Project no: 269317

nSHIELD

new embedded Systems arcHItecturE for multi-Layer Dependable solutions

Instrument type: Collaborative Project, JTI-CP-ARTEMIS

Priority name: Embedded Systems

D6.3: Prototype integration report

Due date of deliverable: M22 –2013.06.30

Actual submission date: M27 – 2013.11.06

Start date of project: 01/09/2011 Duration: 36 months

Organisation name of lead contractor for this deliverable:

Hellenic Aerospace Industry, HAI

Revision [Final]

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2012)		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	X
CO	Confidential, only for members of the consortium (including the Commission Services)	

Document Authors and Approvals

Authors		Date	Signature
Name	**Company**		
Kiriakos Georgouleas	HAI	24/07/13	
Nikos Pappas	HAI	24/07/13	
Balazs Berkes	S-LAB	20/09/13	
Mariana Esposito	ASTS	23/09/13	
Francesco Flammini	ASTS	23/09/13	
Lorena de Celis	AT	24/09/13	
Carlo Pompili	TELC	24/09/13	
Antonio Bruscino	SESM	07/10/13	
Ester Artieda Puyal	INDRA	08/10/13	
Andrea Morgagni	SES	23/10/13	
Luca Geretti	UNIUD	30/10/13	
Tor O Steine	ALFATROLL	31/10/13	
George Hatzivasilis	TUC	31/10/13	
Konstantinos Fysarakis	TUC	31/10/13	
Konstantinos Rantos	TUC	31/10/13	
Alexandros Papanikolaou	TUC	31/10/13	
Harry Manifavas	TUC	31/10/13	
Andreas Papalambrou	ATHENA	31/10/13	
Panagiotis Soufrilas	ATHENA	31/10/13	
Dimitrios Stachoulis	ATHENA	31/10/13	
Vasilios Siadimas	ATHENA	31/10/13	
Kyriakos Stefanidis	ATHENA	31/10/13	
Paolo Azzoni	ETH	31/10/13	
Stefano Gosetti	ETH	31/10/13	
Inaki Eguia	TECNALIA	31/10/13	
Dimitris Geneiatakis	TUC	31/10/13	
Andrea Fiaschetti	UNIROMA1	31/10/13	
Roberto Cusani	UNIROMA1	31/10/13	
Gaetano Scarano	UNIROMA1	31/10/13	
Andrea Morgagni	SES	31/10/13	
Reviewed by			
Name	**Company**		
Approved by			
Name	**Company**		

Applicable Documents		
ID	**Document**	**Description**
[01]	TA	nSHIELD Technical Annex

Modification History		
Issue	**Date**	**Description**
V0.1	24.07.2013	ToC, Introduction
V0.2	26.07.2013	Integration Methodology
V0.3	20.09.2013	Middleware IDS prototype (S-LAB), Additions in Railway Scenario (ASTS)
V0.4	23.09.2013	Prototype oPACKS (AT, TELC)
V0.5	07.10.2013	Gateway (SESM), Link layer security (INDRA), Network Layer Security, Automatic Access Control, Policy Based Management (TUC)
V0.6	24.10.2013	Attack Surface Metrics, Protection Profile (SES), Integration approach, Reputation based Secure Routing (HAI)
V0.7	31.10.2013	Dependable Distributed Computation Framework (UNIUD), Reliable Avionic (ALFATROLL), Reputation based Secure Routing, Automatic Access Control, Policy based management (TUC), Key exchange protocol, Recognizing DoS, Adaptation of legacy systems (ATHENA), Executive summary (HAI)
V0.8	04.11.2013	Middleware Prototypes (UNIROMA1, SES), Metrics approach (TECNALIA)
Final	06.11.2013	Final review

Executive Summary

D6.3 is the first deliverable of Task 6.1 aiming to present the integration of components and prototypes developed in WP3 (node layer), WP4 (network layer) and WP5 (middleware/overlay layer), the interoperability of the various SPD modules and the addressing of all SPD metrics and requirements that the integrated solution needs to meet. The work of this deliverable presents a general framework compliant with the nSHIELD reference architecture that can be used to construct a composable system build on different SPD components from the node, network and middleware components. The integration aspects of the four application scenarios, namely railways security, voice/facial recognition, reliable avionic and social mobility are covered in detail connecting integration activities with the real application demonstrators developed within the project.

Contents

Figures

Tables

Glossary

Please refer to the Glossary document, which is common for all the deliverables in nSHIELD.

This Page is intentionally left blank

1 Introduction

This document is the first integration report inside WP6, “Platform Integration, Validation and Demonstration”. Its purpose is to explore system composability based on different prototypes developed for node, network and middleware layers, concentrate the prototypes needed for each nSHIELD application, explore the interoperability and interdependencies of these and proceed in a first stage of prototypes integration. A second deliverable update will follow, the objective of which is to unify the integral nSHIELD software and hardware units, demonstrating the features of an (as much as possible) integrated platform.

The integration process is connected with the totality (or at least the basic milestones) of the up to now conducted studies and developments. This is a dynamic correlation having the nature of background, bidirectional feedback and on going development. To this date a series of notions have been studied and contribute as cornerstones in nSHIELD integration and outcome, while in parallel these technical topics are under cooperative adjustment and optimization. High level requirements have been defined, imposed by use cases needs and the nSHIELD layers and stratification. Taxonomy of metrics has been presented, along with methodologies to quantify them, in order to better reflect the waviness of SPD levels. The reference architecture has been described, including plain but explicit recitation of the types of nSHIELD devices, layers and functionalities. The definition of interfaces connecting components is the next development step concerning both the finalization of nSHIELD architecture and the integration, demonstration and tuning of nSHIELD platform. In the framework of WP3-WP5, numerous prototypes of node, network and middleware layers are in an advanced or less mature stage of development, whereas the 4 application domains and correspondent scenarios are being built. All these activities will come to synchronize efforts in the integration roadmap. The technical result of what would be a materialized nSHIELD system will be validated and demonstrated to prove (among others) its concept, functionality, reliability, and SPD level compliance and eventually prove its applicability.

The integration task is a complicated one, having a lot to do with available input, compatibility between components, ground conventions and application requirements. The first step would be (not exhaustively) answering questions and defining parameters related with nSHIELD components:

- Node definition and correspondence to working field equipment
- Nodes’ Operating Systems (OS) and their capabilities
- Internetworking options (Gateway based or Direct connectivity)
- Application needs for interoperability
- Middleware and Overlay as the core of system’s structure
- Security policies applied
- Functionalities and capabilities and their trade-off with resource consumption
- Grouping of applications, components and functionalities
- Interfaces (intra-node, intra-layer, between devices, user interfaces)
- Prototype implementation status

The document’s structure begins introducing a review of software methodologies used in the integration of large software systems. Then a system integration approach is presented in Chapter 3, proposing a framework that starting from a list reciting all the developed prototypes and all platforms used as nodes enables a user to select all the SPD algorithms and components running on an integrated system which is compliant with the nSHIELD reference architecture. The integration work is organized on the discrimination of the 4 application scenarios and the components’ distribution, use and integration aspects in these use cases. For each scenario an interaction map is drawn to result in the integrated subsystem that will meet the specifications of the respective application.

2 Software Integration Methodologies

2.1 Introduction

Integration is a critical phase in most cases of software development process as the discrete subsystems developed during the implementation phase must be tied together to bring the required functionalities that the system aims to target. As other activities of software development integration phase is greatly affected by project size. As the project size increases, the number of involved persons and the need for formal communication increases and the portion of time for all the kinds of activities a project needs can change dramatically. A representative figure taken after examining numerous projects of different sizes that shows the proportions of different activities for projects of different sizes is presented in Figure 2-1. It is obvious that for large systems despite how carefully the construction phase have been accomplished, integration and system testing together will dominate the percentage of the total time needed for a software project to be completed and their rate of increase is more that linear.

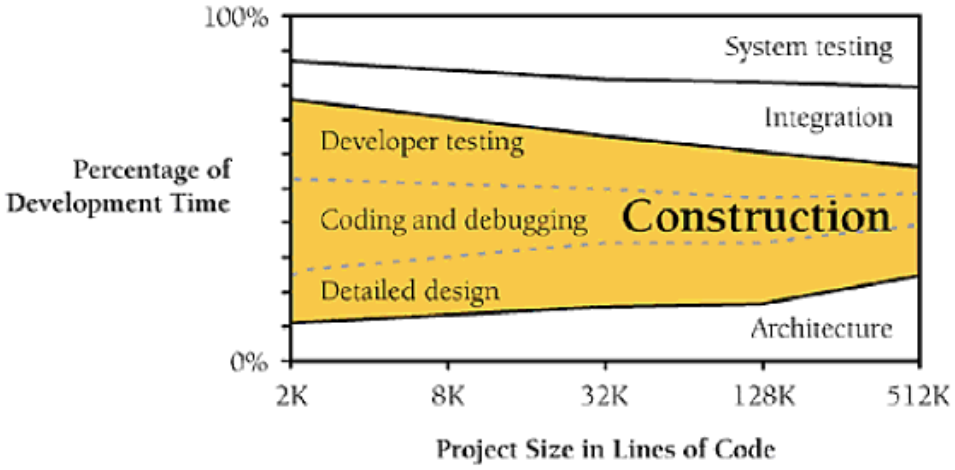

Figure 2-1: Percentage of Development time for different Project Sizes in Lines of Code

As it can be seen by the figure, larger projects require more architecture, integration work and system testing to succeed. Requirements work is not shown on this diagram because requirements effort is not as directly a function of program size as other activities are.

Software integration is performed after finishing developer testing and in conjunction with system testing. For this reason it is sometimes regarded as a testing activity but as the size and the complexity of the target system increases it becomes an independent activity. Key benefits expected from high quality integration are:

- Fewer defects
- Easier defect diagnosis
- Shorter time to first working product
- Shorter overall development schedules
- Better customer relations
- Improved morale
- More reliable schedule estimates
- More accurate status reporting
- Improved code quality
- Improved chance of achieving full functionality of the system under development

2.2 Software Integration Approaches

Over the time, along with different software development life-cycle (SDLC) models, different approaches of software integration have been presented [1]. In this section a short presentation of the most important ones is performed aiming to select or adapt a proper solution for the nSHIELD needs. It is obvious that integration which will include different prototypes developed during the WP3, WP4 and WP5 of the nSHIELD will have a central focus on real application demonstrators integrating SPD modules according to the needs of each of them.

2.2.1 Phased / Incremental Integration

In a first distinction programs are integrated by means of either phased or incremental approach. Phased integration which was the dominant approach until a few years ago, postponed integration after each individual subsystem was completed. It follows these well-defined steps:

- Design, code, test and debug each software module (unit development)
- Combine the modules into the big system (system integration)
- Test and debug the whole system

One problem with this approach (called also Bing Bang integration) is that it examines interoperability of {a large number of} modules that have never worked together before at a late phase increasing the possibilities for a large number of errors to appear and making harder code debugging. An alternative approach is incremental integration where a program is written and tested in small pieces combined one at a time. The steps followed in this approach are:

1. Development of a small functional part of the system that acts as a skeleton where the remaining parts of the system will be attached
2. Design, code, test and debug a module
3. Integrate this module with the skeleton, test and debug the combination of skeleton and the newly developed module. Repeat the process starting at step 2, after making sure that the combination works as expected

Some of the advantages that the incremental approach offers over the phased approach are:

- Easy finding of the location of errors
- Improved progress monitoring
- Extended unit testing and approval that the unit fulfils its goals
- System’s building with a shorter development schedule

With incremental integration the order in which components are constructed is important as it affects the features and functionality of the integrated product over time and careful planning is needed in order the final product to succeed. Different integration-order strategies have been developed that come in a variety of shapes and sizes and none of them is best in every case. The integration strategy must be thoroughly examined to meet the specific demands of the given project.

2.2.2 Top-Down Integration

In top-down integration, the component at the top of the hierarchy is written and integrated first. This top level component can be the portion with the application’s control loop from which lower layer components are called. Stubs have to be written to exercise this top level component. Then, as classes are integrated from the top down, stub components are replaced with real ones. The order of integration for the Top-Down approach is depicted in Figure 2-2.

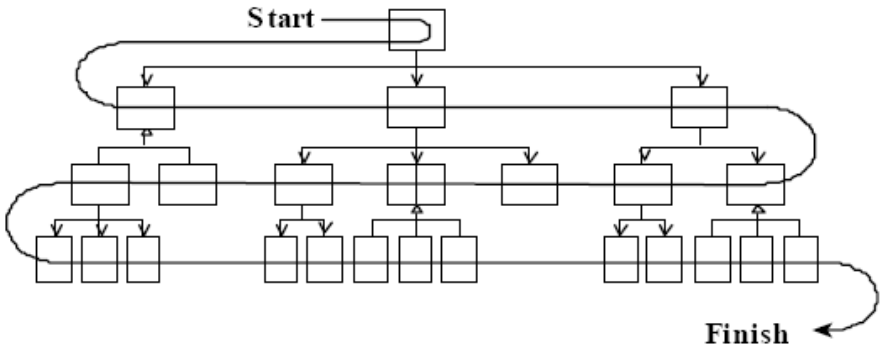

Figure 2-2: Top-Down Incremental integration

In addition to the advantages that included in any kind of incremental integration, an advantage of top-down integration is that the control logic of the system is tested relatively early. Big, conceptual design problems are exposed quickly and the backbone of the system is thoroughly tested over time. Another advantage of the top-down integration is that with careful planning a partially working system can be completed early in the project. Moreover with the Top-down approach coding can begin before the low-level design details are complete.

Pure top-down integration involves a number of disadvantages as well: a major problem is that tricky system interfaces for the full functional system may be not integrated until last. It is not unusual for a low-level problem to propagate its erroneous behaviour to the top of the system causing high-level changes and reducing the benefit of earlier integration work. Stubs developed within this approach are also more likely to contain errors than the more carefully designed production code. Pure Top-down integration maybe also be impossible to be implemented in some systems (meaning that all the higher level components have to be integrated before proceeding with the first lower level component) due to functional reasons so most people use a hybrid approach such as integrating from the top down in sections instead. In this approach (Figure 2-3) the system is implemented in sections fleshing out areas of functionality one by one, and the moving to the next area.

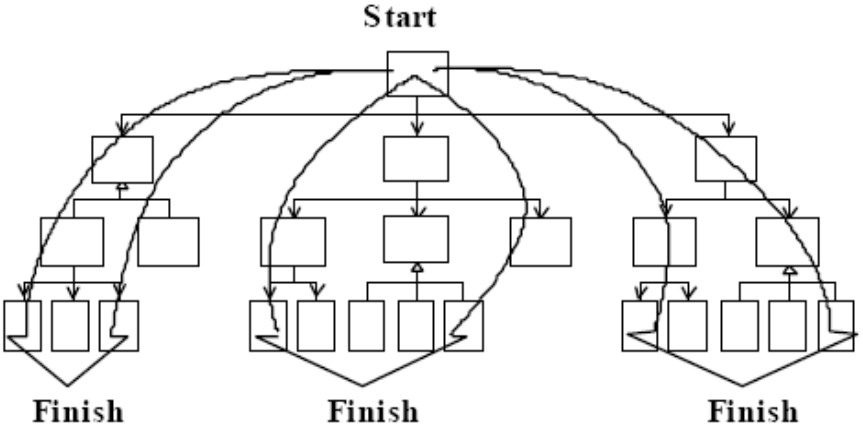

Figure 2-3: System integration from the top down in vertical slices

2.2.3 Bottom-Up Integration

In bottom-up integration, components are written and integrated at the bottom of the hierarchy first (Figure 2-4). Test drivers are used to exercise the low-level components initially and subsequently higher level classes are added to the test-driver scaffolding as they are developed.

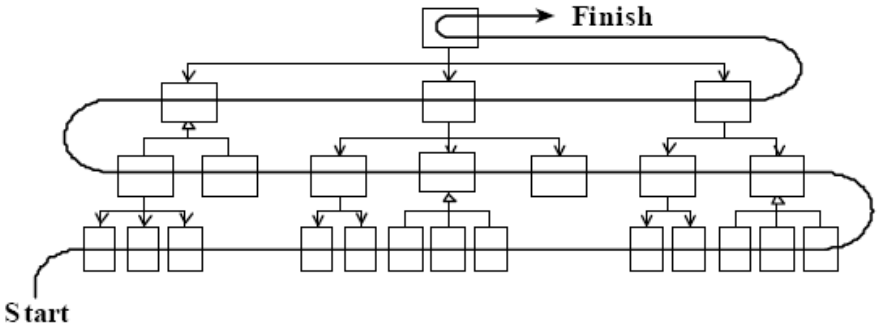

Figure 2-4: Bottom-up integration

Bottom-up integration provides a limited set of incremental integration advantages. Its main advantages are the easy detection of error location and the early integration of full functional low level components early in the project. On the other hand leaving integration of the major high-level system interfaces until last can lead to serious problems as conceptual design problems at the higher levels will not have been identified. Significant changes in design will have as a result to discard some of the low-level work already developed. This method also requires a complete design of the whole system before the start of the integration. Pure bottom-up integration is rare and a more applicable alternative is the hybrid approach of integrating bottom-up in sections in a way similar to this of top-down in vertical slices approach.

2.2.4 Other Integration Approaches

As pure top-down and bottom-up integration are difficult to be applied in most real cases a number of integration alternatives have been proposed.

In Sandwich integration [2], the integration starts from the high-level business-object classes at the top of the hierarchy and continuous with low level device-interface classes and widely used utility classes of the lower level. Middle-level classes are left for the end of integration.

In risk-oriented integration each system component is associated with a level of risk, and the system is integrated with the most challenging parts to implement first. The remainder of the code which is easier to implement is left for integration later. An illustration of risk-oriented integration is presented in Figure 2-5.

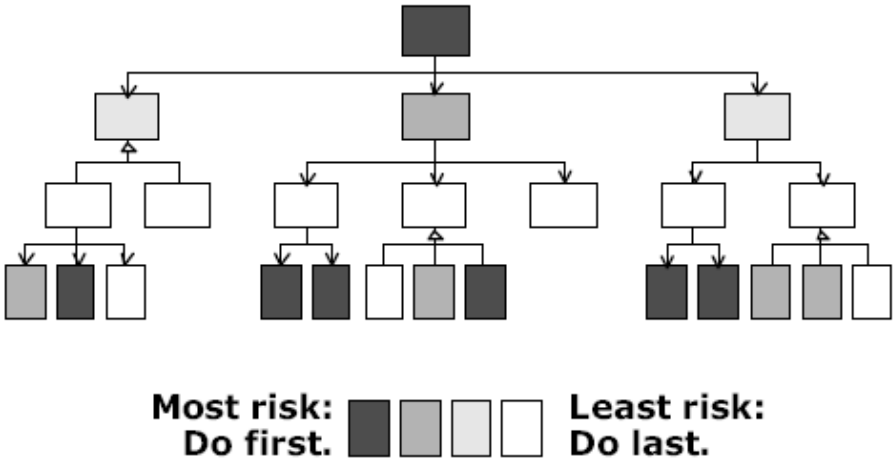

Figure 2-5: Risk-oriented integration

In feature-oriented integration the key focus of integration is a functional feature (e.g. automatic reformatting of a document in a word processor) that may expand to more than one single component. A

prerequisite for this method is the existence of a skeleton upon which the new features will be integrated and tested. Components are usually added in “feature trees”, hierarchical collections of classes that make up a feature. An illustration of feature-oriented integration is presented in Figure 2-6.

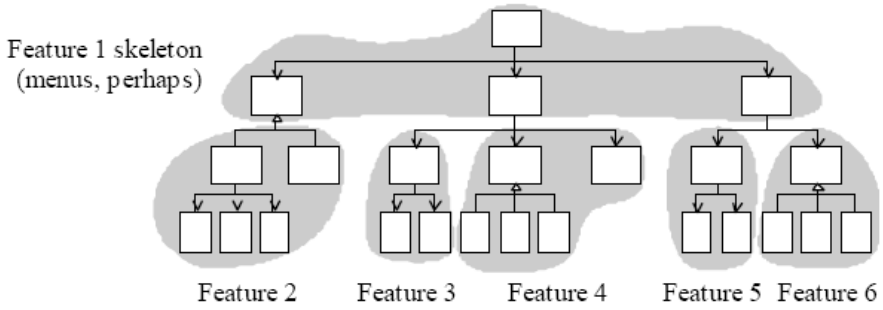

Figure 2-6: Feature-oriented integration

A final approach that often addresses the problems associated with top-down and bottom-up integration is “T-Shaped Integration” (Figure 2-7). In this approach, one specific vertical slice is selected for early development and integration. This slice exercises the system end-to-end and should be capable to reveal any major problems in the system’s design assumptions. Once the implementation and testing of a vertical slice has been completed the procedure continues with the next slice until the whole system is finished. This approach can also be combined with risk-oriented or feature-oriented integration.

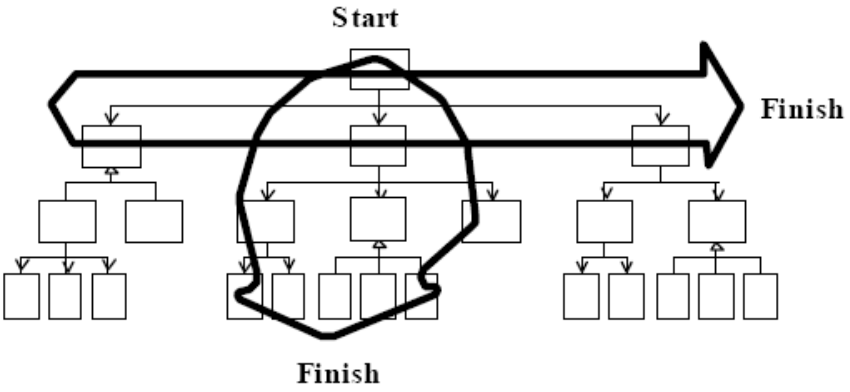

Figure 2-7: T-Shaped integration

2.3 Version Control Systems

Along with Incremental Integration, Source Code Version Control is another major approach to gain in productivity and increase quality and consistency when constructing advanced software systems. Source Code Control is essential to allow software development teams to work effectively together and let each team member to make progress without interfering with the work of other people of the team. Some of the big benefits of incorporating version control in software development are:

- Parallel working on a file while someone else is working on it
- Easy update of all project’s files to the latest version usually by issuing a single command
- Backtrack to any version of any file that was ever checked into version control
- Ability to track the list of the changes made to any version of any file
- Avoidance of the burden of personal backups as the version control system works as a safety net

Particularly in integration version control can provide a safe backup point with branching, where a new branch is constructed each time a new subsystem is being integrated. In case a problem appears in the integration of a new feature rework can easily restart at the point of the previous branch containing all the work prior the attempt to integrate the new feature. The degree of help that a version control system provides is also affected by parameters like:

- The structure of the development team: A two person team in one room has different needs than a large team spread across the globe.
- The software product architecture: Some points in the architecture allow for a greater degree of decoupling than other points.
- The capabilities of the version control tool: The selected tool must support effectively all the features required from a version control tool for the specific product.

All these parameters must be thoroughly examined prior the adoption of a version control tool to maximize the possibilities of a successful integrated product developed on time.

2.3.1 nSHIELD SVN Repositories

For the purposes of nSHIELD project Apache Subversion [15] was used as revision control system for software versioning. A SVN first repository managed by MGEP for source code integration of WP4 activities has been deployed at:

```
svn://forja.mondragon.edu/scmrepos/svn/nshieldwp4/
```

A second SVN repository related to WP5 activities managed by UNIROMA has been deployed at:

```
svn://labreti30.ing.uniroma1.it/nshield/code
```

2.4 Software Integration Checklist

The following list of activities is provided as a tool that helps in clarification of prerequisites for successful integration as well as in the adoption of the strategy that serves better the project needs.

1. Identification of the optimal order in which subsystems, classes and routines should be integrated.
2. Examination of the relevance between integration order and construction order so that components are ready for integration at the right time.
3. Examination for well-defined interfaces between components
4. Adoption of an integration strategy (or combination of strategies) that offer benefits over other alternative strategies. Key attributes in which each integration strategy should be examined include:
    a. Easy debugging and diagnosis of defects
    b. Minimal development of supplementary code for integration only purposes
    c. Easy integration of new components

3 nSHIELD Integration Approach

3.1 Overall approach

The aim of the Multi-Technology system integration task it to compose seamlessly components and prototypes developed in WP3, WP4 and WP5 in order to address all SPD concerns and requirements of a real application scenario. As described in the nSHIELD Technical Annex [3] composability and architectural dependability are two priorities of highest importance for the nSHIELD project.

After selecting the most appropriate SPD algorithms, technologies and procedures and developing the missing ones, nSHIELD must be capable to integrate and harmonize them in a modular, composable, expandable and high-dependable architectural framework. A first step toward this goal is to identify the framework inputs that will make the composable system a reality. Figure 3-1 helps in the clarification of the required inputs and the produced output for the system integration procedure.

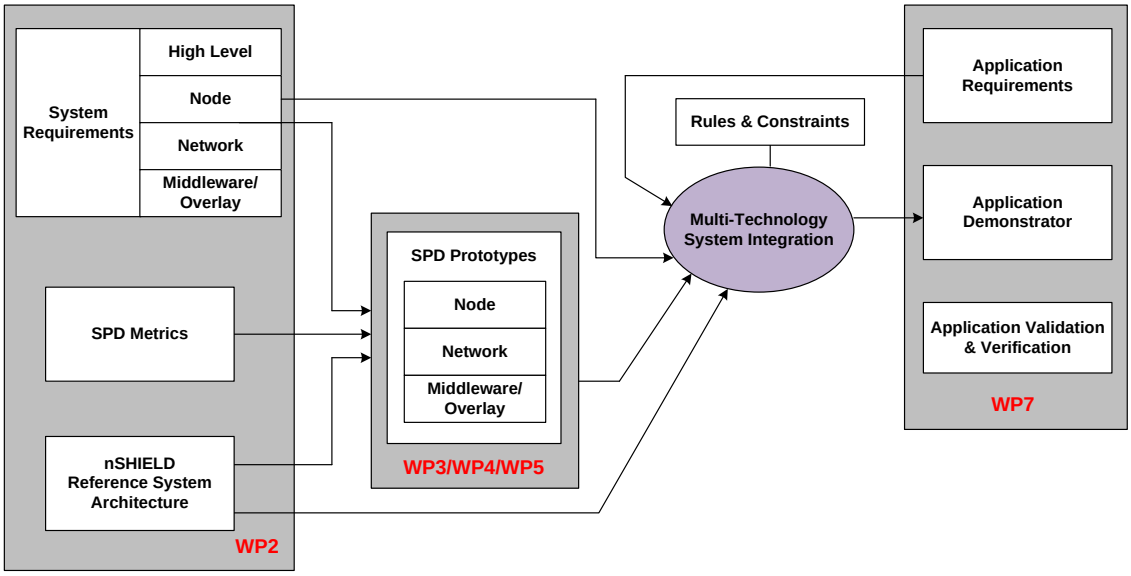

Figure 3-1: Interdependencies between nSHIELD tasks

Starting from the system requirements presented in [4] where both high level requirements related to nSHIELD application scenarios and requirements related to each one of the nSHIELD layer (node, network, middleware, overlay), the identification of SPD metrics in [5] and the overall reference architecture design [6] where all types of nSHIELD embedded devices and means of their interconnection where identified, a number of prototypes related to node [7], network [9] and middleware/overlay [11] layer were constructed. The prototypes list of Table 3-1 contains all technology prototypes developed within nSHIELD activities.

Having available the list of prototypes the first step of the System Integration is to analyse the application requirements and crosscheck them with the repository of system requirements to find the degree of fulfilment of both application and SPD requirements. Considering nSHIELD an open and expandable security and dependability framework, the number of SPD prototypes will be increased in the future covering new aspects as technology evolves. After the assessment of requirements fulfilment and in the case of a positive result application requirements must be decomposed to node, network and middleware/overlay parts. At node layer the framework must be able to distinguish the network interface for interconnected node, the inclusion of nS-ESD Gateways for supporting legacy and proprietary networks as well as to evaluate the SPD capabilities of each node. At network layer the framework must be able to provide efficient communication among all the involved entities meeting the requirements set by

the specific application. The middleware requirements must be also able to be supported from the nodes (constrained devices may not allowed to have a direct connectivity to middleware services) otherwise the system will may be functional composable but with lower SPD capabilities. The framework must be able to provide a general SPD level assessment as well as a list of threats addressed and possible known vulnerabilities before application composition.

Table 3-1: nSHIELD prototypes

PROTOTYPE LIST		
00	Elliptic Curve Cryptography	UNIGE
01	Lightweight Ciphering	TUC
02	Key Exchange Protocol	TUC
03	Hypervisor	SICS
04	Secure Boot	T2D
05	Secure Power (&) Communication cape	AT
06	Smart Card based Security Services	TUC
07	Facial Recognition	ETH
08	GPU accelerated Hashing	TUC
09	Smart Transmission	SES/UNIGE
10	Anonymity & Location Privacy Service	TUC
11	Automatic Access Control	TUC
12	DDoS Attack Mitigation	ATHENA
13	Recognizing DoS	ATHENA
14	Dependable Distributed Computation Framework	UNIUD
15	Intrusion Detection System	MGEP
16	Reputation-Based Secure Routing	TUC/HAI
17	Access Control Smart Grid	TECNALIA
18	Policy Definition	ASTS/SES/SESM
19	Policy Based Management Framework	TUC/HAI
20	Control Algorithms	UNIROMA
21	Gateway	SESM
22	Middleware Intrusion Detection System	S-LAB
23	Link Layer Security	INDRA
24	Network Layer Security	TUC
25	OSGI Middleware	UNIROMA1
26	Semantic Model	UNIROMA1
27	Multimetrics	TECNALIA
28	Attack Surface Metrics	SES
29	Adaptation of Legacy System	ATHENA
30	Reliable Avionic	ALFATROLL
31	Middleware Protection Profile	SES
32	Secure Discovery	UNIROMA1
33	Security Agent	UNIROMA1
34	Audio Surveillance System	ISD
35	Beagle Board-Xm	SICS
36	OMNIA-IMA	SES
37	ETH SecuBoard	

Prototypes of Table 3-1 can be distinguished according to the layer designed for as presented in Figure 3-2.

Node Prototypes	
00	Elliptic Curve Cryptography
01	Lightweight Ciphering
02	Key Exchange Protocol
03	Hypervisor
04	Secure Boot
05	Secure Power (&) Communication Cape
06	Smart Card based Security Services
08	GPU-accelerated Hashing
10	Anonymity & Location Privacy
11	Automatic Access Control
07	Face Recognition
34	Audio Surveillance System
30	Reliable Avionics

Network Prototypes	
09	Smart Transmission
12	DDoS Attack Mitigation
13	Recognizing DoS
14	Dependable Distributed Computational Framework
15	Intrusion Detection System
16	Reputation-based Secure Routing
17	Access Control Smart Grid
23	Link Layer Security
24	Network Layer Security

Middleware/Overlay Prototypes	
25	OSGI Middleware
26	Semantic Model
18	Policy Definition
19	Policy Based Management Framework
20	Control Algorithms
21	Gateway
22	Middleware Intrusion Detection System
29	Legacy System Adapter
31	Middleware Protection Profile
32	Secure Discovery
33	Security Agent

Figure 3-2: nSHIELD prototypes at node, network and middleware/overlay layer

3.2 System composition

The enormous amount of Security, Privacy and Dependability features covered by nSHIELD prototypes poses a lot of challenges to system composability. Integration of different prototypes will be firstly driven by application scenario requirements where security and dependability requirements should also be expressed. It is neither feasible nor required for a scenario to include all the prototypes of Table 3-1 as different applications have different requirements and some essential prototypes for one scenario may be meaningless for another. The following section provides a guide for prototypes composition which can form the basis for the creation of a software framework that working together with SPD metrics quantification software will help in the integration of the different technologies developed throughout the nSHIELD project as well as in the SPD level assessment compliant with the work presented in [5]. At this point prototypes of Table 3-1 are focused on a specific SPD feature and their initial implementation has been performed in isolation from the other prototypes. The final version of the integration report will receive feedback from the attempts to compose systems that include different prototypes in four application scenarios, railway security, voice/facial recognition, dependable avionic and social mobility and networking.

The first step performed by the composition framework is the selection of the most appropriate hardware nodes from the repository of all valid nSHIELD platforms. This repository will include all platforms presented in Table 3-2 expanded with new platforms while nSHIELD project evolves

Table 3-2: nSHIELD SPD nodes used during prototypes development

nSHIELD SPD node	Operating System	Application Processor (Legacy Device Component)	Non-volatile memory (Legacy Device Component)	Volatile memory (Legacy Device Component)	Special-Purpose Processor (Legacy Device Component)	Stable Storage (Dependability Block)	I/O Interface	Power Management	Waveform / Freq. Band / Data rate
Zolertia Z1	Contiki / TinyOS	MPU @ 16MHz	EEPROM CP2102 usb-to-uart (Please check)	RAM 8KB	12 bit ADC	92KB Flash or 16MB (Please check)	USB, UART, GPIO, 802.15.4 (ZigBee) (CC2420 transceiver)	Batteries / USB	DSSS / 2.4 Ghz / 250 kbps
Raspberry Pi	Linux	ARM11 @ 700MHz + GPU	-	SDRAM 256 MiB (mebibyte)	DSP available but not currently public API	Secure digital SD/MMC/SDIO card slot (4GB)	USB, UART, GPIO, Ethernet	Micro USB / GPIO	No wireless module
Arduino Uno	Arduino software	MCU @ 16 MHz	1KB EEPROM	SRAM 2KB	-	32KB Flash	USB, UART	Batteries / USB	WiFi (802.11 b/g) and Wireless (802.15.4) Shields available
BeagleBoard	Linux	MPU @ 720 MHz (OMAP)		DDR2 256MB		SD/MMC/SDIO card slot	USB, UART, GPIO	USB / DC	No wireless module (interface available)
BeagleBoard -XM	Linux / WinCE	DSP @ 1 GHz (DM3730)		DDR2 512MB		microSD card slot	USB, UART, GPIO	USB / DC	No wireless module (interface available)
BeagleBone	Linux	MPU @ 720 MHz (DC powered) OR MPU @ 500 MHz (USB powered)	32KB EEPROM	DDR2 256MB		microSD card slot (4GB)	USB, UART, GPIO, I2C, Ethernet	USB / DC	WiFi (802.11 b/g) and Wireless (802.15.4) Capes available

OMBRA v2	Linux / WinCE	ARM Cortex CPU @ 1GHz		LPDDR 1GB	DSP & FPGA	microSD (32GB)	USB, Ethernet, VGA	Usb / DC	No wireless module at the moment (future versions may include RF front end)
Memsic IRIS	TinyOS / Contiki (port)	Atmel ATMega 1281, 8MHz	4KB EEPROM	RAM 8KB	10bit ADC	128KB Program Flash Memory, 512KB Measurement (Serial) Flash	UART, I2C, SPI	Batteries (2xAA)	802.15.4 (2.4 Ghz / 250 kbps)
Eurotech SecuBoard	Linux WinCE (optional)	ARM® Cortex-A8 RISC Processor (up to1,35 GHz)		1 GB DDR3 @400MHz	VLIW floating-point DSP core - 3 Programmable HD Video Image Coprocessing (HDVICP2) Engines, 3D Graphics Engine, Integrated CMOS sensor (M031 or T001 or ICD445 or AR0031 FullHD)	512 MB Nand Flash / SD Card	USB, Ethernet (1Gb), UART, GPIO, SPI, IrDA, CIR, I2C, eSATA, PCIe, McBSP, HDMI, APC for FPGA,CPLD and ASICS	DC (9-15 Vcc), PoE (optional) Batteries (optional)	WiFi, 3G

Application requirements will guide the selection of the nodes. The output of this process, Figure 3-3, will be a list of all hardware platforms able to successfully compose the application, a list of all prototypes able to run on them together with an initial SPD level assignment. The analysis of this step should include the following steps:

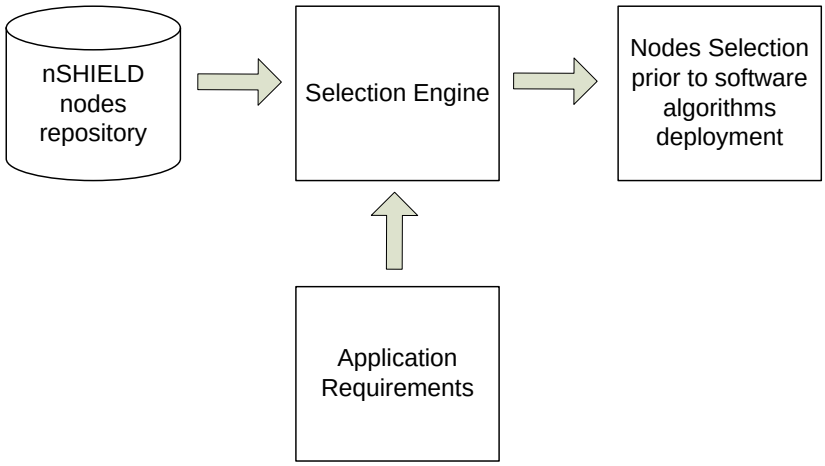

Figure 3-3: Process for selecting hardware platforms prior to SPD algorithms deployment

1. Physical Protection: The necessity of anti-tampering detection, TPM for secure storage of cryptographic keys and secure power unit must be examined during node selection. BeagleBone Secure Power and Communication Cape (Prototype 5) is able to provide physical protection in any application scenario that uses BeagleBone as platform/node.
2. Secure Boot & Execution Environment: Platforms that support secure execution environment that allows trustworthy, security critical applications to run isolated from other co-existing less trustworthy/insecure applications provide another input for the SPD level assessment. BeagleBone platform supports these features implemented in Prototypes 3 & 4.
3. Operating System: Depending on their resources nSHIELD nodes can run on a variety of operating systems from more full-featured like embedded Linux and Android to lightweight like Contiki and TinyOS. Security and dependability features of the operating system must be evaluated.
4. Network interfaces: Network requirements must be analysed for each scenario and the selected node must support this network interface which can be either wired like Ethernet, USB or serial interface or wireless like SDR, 802.11, 802.15.4, RFID, Smart Card, Bluetooth etc. The network interface will form the base for the examination of network layer security and dependability algorithms applicability.
5. Java Runtime Environment: Nodes able to support core nSHIELD SPD services denoted as ns-SPD-ESD nodes in [6] must be equipped with Java Runtime Environment for dynamic SPD services establishment.
6. Hardware Accelerators/Specific processors: The need of using specific hardware accelerators/processors to run at node a required prototype must be examined at this step. Prototype 8 GPU-accelerated hashing belongs to this category.

An example template with SPD features and supporting functionality prior node/network/middleware SPD algorithms deployment is provided in Table 3-3.

Table 3-3: Example template of HW platform with HW SPD features, available network connectivity and algorithms ready to run

Platform Name: BeagleBone	Feature Supported	
	Yes	No
Physical Protection		
Custom Encapsulation	X	
TPM Crypto-coprocessor	X	
Secure power unit	X	
Secure Execution Environment	X	
Secure Boot	X	
Operating System		
Embedded Linux	X	
Communication Interface		
Ethernet	X	
USB	X	
Serial	X (add-on cape)	
SDR		X
802.11	X (add-on cape)	
802.15.4	X (add-on cape)	
RFID		
Smart Card		
Bluetooth		
Java Runtime Environment	X	
GPU Unit		X

Prototypes able to run					
Node Layer		**Network Layer**		**Middleware/Overlay Layer**	
00	*Elliptic Curve Cryptography*	24	*Network Layer Security*	19	*Policy Based Management*
01	*Lightweight Ciphering*			32	*Secure Discovery*
02	*Key Exchange Protocol*			33	*Security Agent*
10	*Anonymity and Location Privacy*				
11	*Automatic Access Control*				

The next step where the framework will be utilized in system composition is the selection of the SPD algorithms that will run at each platform taken from node/layer/middleware prototypes. A distinction at this point must be provided between features that needed to be selected statically (at design time and prior to node programming and deployment) and those that can be configured at runtime which are closely related to middleware services. Source code integration of prototypes with related functionality can follow the

general procedures described in Chapter 2. The framework must be also able to resolve conflicts and solve issues related to the involvement of legacy and proprietary nodes where some kind of Gateway (nS-ESD Gateway in the nSHIELD reference architecture [6] addressed from prototype 21) and Legacy System Adapter (addressed from prototype 29) must be used. After system composition the system administrator must have a clear view of all SPD components included in each node, an interface to interact with runtime parameters as well as SPD levels at node, network and middleware layers for all elements included in system configuration (Figure 3-4). For more complicated and numerous systems a clustering approach will be followed. A more detailed description of composability framework will be presented in D6.5.

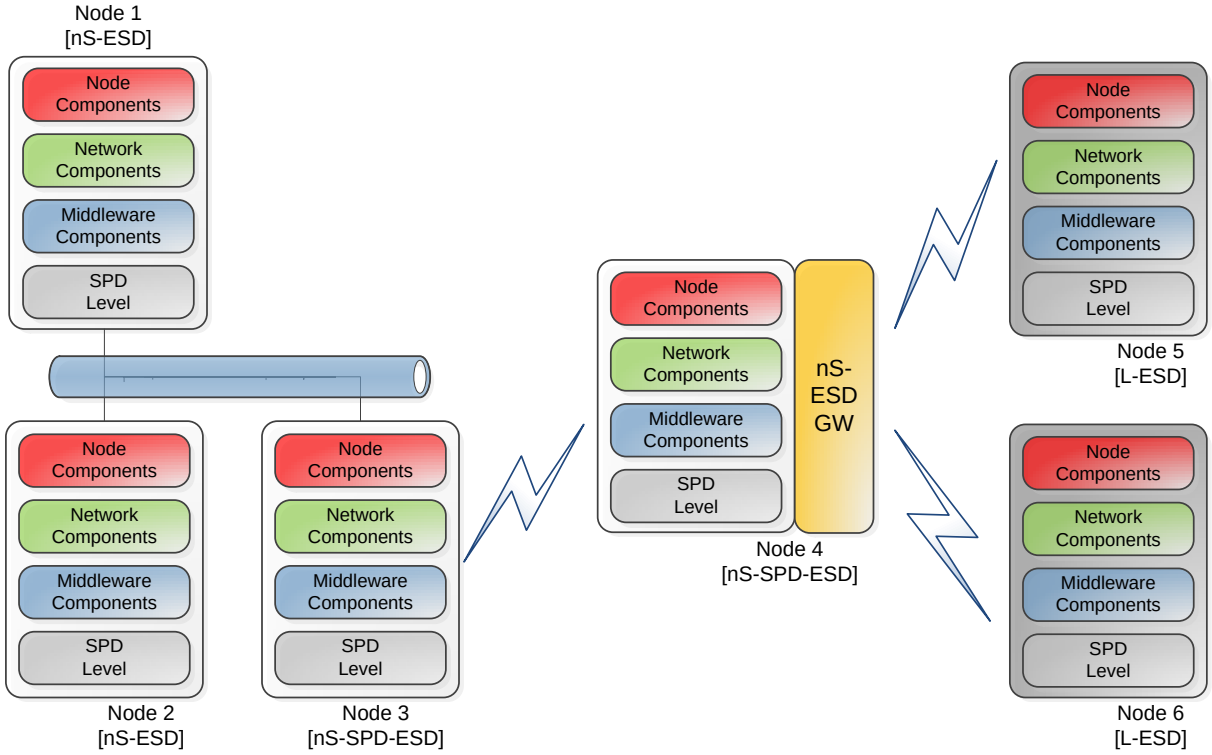

Figure 3-4: Composed nSHIELD system: information about node, network and middleware components includes SPD level assessment for each system element

The above picture can be compared with Figure 3-5 taken from the reference architecture [6]. The view of an integrated and working nSHIELD system must be compliant with the conceptual architecture giving in their users and administrators the ability to have a clear picture of the SPD algorithms implemented in each network element, a quantified value of the security/dependability applied as well as the ability to change dynamically at run time the SPD level utilizing middleware services deployed.

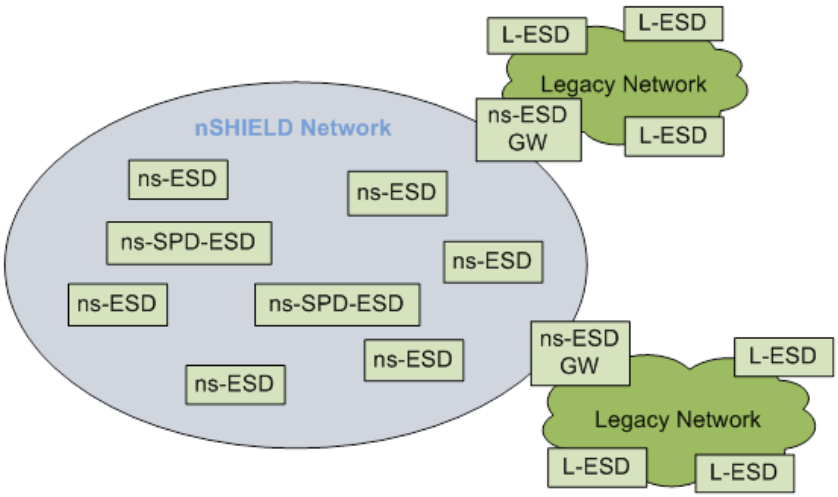

Figure 3-5: Conceptual Architecture of the nSHIELD system

4 Integration of Railway Scenario Components

This chapter presents the description of different prototypes used in railway scenario. Components of the middleware layer have been built with the adoption of the Knopflerfish OSGi open source service platform which facilitates the integration process. The chapter concludes with an interaction map that presents the interfaces and interactions needed to build the railway security system from all the components that participate in system composition. A more detailed picture of interactions among the involved components will be presented in the final version of this deliverable taking feedback from the development effort of the railway security demonstrator.

4.1 Control Algorithms (Prototype 20)

The SHIELD control algorithms are the simplest prototype to be integrated in the common platform, since they are embedded in the Middleware code and are implemented in the Security Agent bundle. On a practical point of view, the control algorithms will be a set of software instructions that will implement the SPD driven composition as described in D5.2 and D5.3 as follows:

- The OSGI services will populate the knowledge bases that contain the semantic representation of the SHIELD components and the scenario dependent information
- This information will be merged by means of adequate control algorithms to compute the list of SPD functionalities that have to be activated to satisfy the user requirements in terms of SPD.
- This list will be forwarded to the middleware to propagate the control command to system components

The strength of this solution is that the behaviour of the control algorithm module is independent from the specific control algorithm implemented, leaving the possibility of exploring more than one solution within the same framework (if needed).

In addition, in case the OSGI libraries will not be suitable to solve the composition (mathematical) problem, then the support of external computational software tool can be foreseen, like, for example, Matlab, that could be easily called by the Security Agent routines to solve the problems.

4.2 Middleware Intrusion Detection System (Prototype 22)

4.2.1 IDS prototype interfaces

In its current status, the preliminary IDS prototype has generic network interfaces for receiving and forwarding requests that are to be filtered. In Figure 4-7, these bi-directional interfaces are denoted as **IF-2**. The IDS prototype will receive and optionally forward messages without altering their content or re-encapsulating them. In this sense, **IF-2** is a homogeneous interface between the Middleware services and the prototypes which use them.

It is however anticipated that TAP / TUN virtual network interfaces could be used to physically separate and protect internal (Middleware services) and external (other components and networks besides Middleware) network domains. These changes could mostly be implemented in a transparent manner for the system components using middleware services, but may impact how connection methods towards middleware services should be implemented. The design of the network domains and the connection methods used will be studied at the time of integration with other Middleware components.

Using the IDS prototype requires setting up network infrastructure so that requests are received by the gateway instead of the middleware services natively. For this purpose, the Intrusion Detection and Filtering Module provides additional function call interfaces towards the Middleware services that implement the use of the IDS – see **IF-3** in Figure 4-7. At the time of integration, it needs to be determined which Middleware services are to be protected against DoS/DDoS attacks, and which operation mode of intrusion detection (blacklisting / whitelisting) is more beneficial to be used for each. Use of IDS prototype

for these services is straightforward: by adding a few lines of Java code to the services, they can be enabled to use intrusion detection functionality.

The nSHIELD Overlay functionality will be responsible to monitor SPD properties and control desired SPD level for the prototypes. The function interface IF-3 provides all functions for this purpose; for more details, see next chapter.

Further information and source code for the IDS prototype is available in D5.3 [11] and D5.2 [10].

4.2.2 IDS prototype SPD features

The preliminary version of the Intrusion Detection and Filtering Module provides the following features for the Middleware and Overlay services utilizing and controlling the IDS. These features are accessible via Java function interfaces in the OSGi Middleware environment:

- Intrusion detection configurable per service
- Provides manual addition and removal of blacklist and whitelist elements for clients – operation mode and lists can be controlled from the Overlay based on higher level semantic SPD information (e.g. based on trust level associated with clients obtained from Secure Discovery)
- Critical Load Detection of Server
- Can be switched to whitelisted or blacklisted mode, or can switch automatically under critical load (can be controlled according to required SPD level changes as well)
- Provides function interface to query Service Metrics that can be used to assess SPD level of the prototype:
    - totalIncomingRequestCount
    - totalOutgoingResponseCount
    - totalDroppedFromQueueCount
    - currentQueueSize
    - totalBlacklistRejection
    - totalWhitelistRejection

4.2.3 IDS prototype environment

The IDS prototype was designed to become part of the Middleware services, thus its function interfaces are implemented in Java, wrapped as a Knopflerfish OSGI Bundle. However, to reduce overhead imposed by the IDS forwarding all Middleware requests, the core functionality was implemented in C++ natively. The code was designed to be portable, available for compilation under different OSes – using either Visual Studio 2012 (Windows) or g++ 4.7.3 (Linux / Windows). The code uses portable libraries (Boost). The core functionality can be compiled either as:

- stand-alone executable (providing default IDS settings for services configured via command line or configuration file)
- DLL / so shared library that can be used from Java via JNI wrapper. This method is implemented by the current solution in the OSGI Framework for Middleware, providing also a function interface to other OSGI bundles.

4.3 Reputation based Secure Routing (Prototype 16)

For the integration of reputation-based secure routing prototype in the railway scenario, network interfacing is the first issue needed to be solved. This of course is true in the case that secure routing is running on 802.15.4 wireless network of embedded device as is the case of Prototype 16. From an architectural point of view the kind of network that secure routing prototype runs can be considered a

legacy network and a gateway device (nS-ESD GW) must be included in order to communicate with the L-ESD devices (Figure 4-1).

From a usage perspective, several kinds of sensors can be attached to sensor nodes (Temperature, Vibration, Pressure, Infrared, etc) according to application needs. A translation between the legacy network and the railway security network must be performed for middleware tasks such as:

- Service discovery where the services provided by the sensor network must be identified
- Data gathering of sensed values
- SPD level management where the security and dependability level of each participating node will be configured. In the case of reputation based secure routing prototype this level could adjust the inclusion of indirect trust in trust calculation and the examination of conformance of third party opinions to a statistical distribution mean value in order to exclude bad-mouthing attacks. More details on this can be found in D4.2 [8] and D4.3 [9].

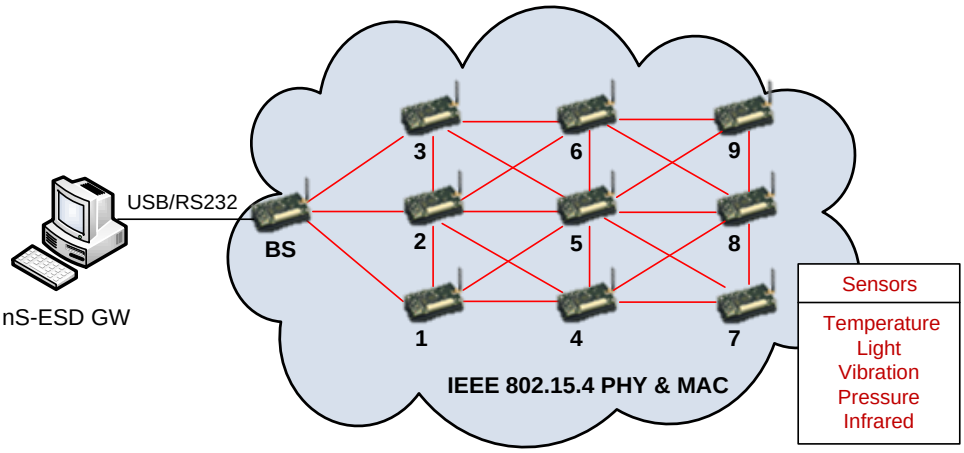

Figure 4-1: WSN secure routing prototype - nS-ESD GW (integration in Railway Scenario)

4.3.1 Reputation-based Secure Routing Prototype Interfaces

WSNs with sensor/mote nodes are applied in railway scenarios. Such nodes are placed along the train's route for continuous monitoring. As it is expensive for all these nodes to be directly connected to the railway station network, they must cooperate to communicate information from one end to the other. Thus, routing protocols can be applied to carry out the communication from nodes to the WSN gateway which is connected to the railway WAN. Due to the difficulty in physically securing all these nodes and avoiding nodes that have been compromised by attackers to interact with the rest of the network, trust and reputation-based schemes for secure routing can be applied. Furthermore, trust and reputation-based routing can act as an intrusion detection mechanism and detect jamming areas.

In Figure 4-7, the general interfaces for interacting with the main railway WAN can be denoted as the prototype "Reputation-Based Secure Routing – 16" in the "Redundant cluster" box as "Other Smart Sensors" technologies. The prototype communicates with the "Network Layer Security / LAN / WAN – 24" component.

Using the reputation-based secure routing prototype requires setting up a WSN, which embodies the proposed trust and reputation-based system, and has at least one gateway that is connected to the Network Layer Security component of the rest nSHIELD railway system. At the initialization phase, the WSN manager must set the parameters of the trust and reputation-based scheme for each node. The nodes will automatically discover the communication routes (using the pure routing protocol procedures). Then the nodes will start performing their main operations (e.g. sense local environment variables). The communications between the sensor nodes are evaluated by the trust and reputation-based system. The

WSN can automatically communicate with the rest railway system to transmit the data (e.g. sensed info, trust and reputation) or arise an alarm in case of attack. The main railway system can make requests to the WSN or change the configuration parameters of the trust and reputation-based scheme at runtime.

The nSHIELD Network functionality will be responsible to monitor SPD properties and control desired SPD level for the prototypes. For more details, see next chapter. Further information and source code for the reputation-based secure routing prototype is available in D4.2 [8] and D4.3 [9].

4.3.2 Reputation-based Secure Routing Prototype SPD features

The preliminary version of the Reputation-Based Secure Routing Module provides the following features for the Network services to configure and control the reputation scheme. The following function interfaces are considered:

- Configuration of the trust and reputation scheme to comply with the application's requirements. A GUI is implemented to ease the configuration process at deployment time.
- Pre-defined options for implementing the decision making process of well-known trust and reputation-based schemes
- Runtime configuration of the scheme to comply with the security policies and the relevant SPD level
- Provides function interface to query Service Metrics that can be used to assess SPD level of the prototype (per node):
    - *averageReputation*
    - *reputationBias*
    - *transactionRate*
    - *agentTypeProfit*

4.3.3 Reputation-based Secure Routing Prototype environment

The reputation-based secure routing prototype was designed to become part of the Network services. Its function interfaces are implemented in C++ and extend the routing protocol DSR. The code is tested under the operating system Linux. The core functionality is implemented in the Linux kernel and rest functionality in the user space. The compilation creates a module in the Linux framework Netfilter, which is responsible for manipulating the out/incoming network traffic of a node.

4.4 Offline Physical Access Control System (Prototype 05)

Railway operators are responsible for a large number of locked assets spread over a large area. Due to the high cost of traditional access control systems, many railway operators still rely on traditional mechanical keys, which are both inefficient and insecure. For example, the operator of the subway system in Stockholm, Sweden, is responsible for more than 20.000 doors, of which only 10% are currently equipped with an access control system.

An offline system is much less costly to install compared to an online system, and therefore an interesting alternative to online systems and traditional keys. Offline Physical Access control solution fits well with the need of a railways operator and can complement the Railways Security System in that areas where a secure online integration present a higher cost.

4.4.1 oPACS prototype interfaces

The nSHIELD partner Telcred (TELC) develops an offline access control system. At the door, the solution is comprised of a reader, a lock controller and an electric lock.

The lock controller is placed on the inside of the door and is a critical component since it is responsible for making the actual access control decisions based on the credentials presented by the user to the reader. In other words, it is highly important that this component is both reliable and resistant to attacks.

Within the scope of the project, Telcred (TELC), Acorde (AT), and SICS are collaborating on developing a secure micro node, which can be used as a lock controller. A custom “cape” for a standard BeagleBone low end Linux computer is being developed. This cape will provide features such as tamper detection, backup power, secure storage of cryptographic keys, and a real time clock.

This system will be used as an offline physical access control and the secure lock controller will operate offline/standalone. This means that, at the beginning, no interfaces with other nSHIELD devices/ components will be implemented.

4.4.2 oPACS prototype SPD features

This nSHIELD node prototype is composed of different subsystems that are directly related to different partners’ expertise. This prototype has been designed as a BeagleBone cape. The main functionalities of this prototype are:

a) **Custom encapsulation + Supervisor and anti-tampering**
b) **Power unit** for the BeagleBone board and third-party boards
c) **TPM module** to support the storage of the security keys that are involved in the partners cryptographic developments. This feature is provided through a holder/slot for a smart card with form factor ID-000 (same as a typical SIM-card). This way, different hardware can be used depending on the application (using a smart card with Java Card for secure storage and to serve as a crypto co-processor).
d) **RF Module** that supports the 802.15.4, based on the MRF24J40 that provides a wireless communication link.
e) **Other features**:
    i. Additional RS-485/RS-232 external interfaces (driver + connector) will be available in the cape.
    ii. RTC signal will be provided.
    iii. Two relays
    iv. Several digital inputs

With this prototype some SPD functionalities that could be covered are listed below:

Table 4-1: Offline Access Control SPD features

Digital Signatures	Different signature verifications can be done using Java Card applets (like implementation of ECDSA java card applet on the smart card)
Physical/tamper resilience	This feature is a requirement that has been considered during design stage. It has been included a supervisor chip to cover this feature connected to a switch.
ECC Authentication	This feature can be covered by means a software solution like using Java Card/JCOP smart cards (built-in functionality in Java Card)
Accommodations for future energy sources	The design of AT custom power module will include a power input interface for alternative power sources and on-board battery to allow the future implementation of power harvesting technologies
Power management	This requirement manages any system power supply risk, which might affect to the node behaviour. In case of failure of any of the countermeasures, being able to protect all the electronics and devices, in order to avoid further damages into the system and increase the node availability

Due these "nSHIELD capabilities" (mainly the crypto implementation) this micro node allows to communicate securely with external devices such as NFC phones, smart cards and wireless sensors.

4.4.3 oPACS prototype environment

For the use case demonstration, the physical access control system can be integrated with other nSHIELD components through the back end. In other words, the overarching SMS (security Management System) can be integrated with the oPACS (Physical Access Control System) on the back end, while the secure lock controller will operate offline/standalone.

4.5 Network layer security (Prototype 24)

4.5.1 Network layer prototype interfaces

The network layer prototype developed can be connected to other systems through a gateway that supports the underlying protocols for offering network layer security.

4.5.2 Network layer prototype SPD features

The network layer security prototype deals with the ability to provide message protection at the network layer. In this way, any sensitive information transmitted among nodes will be secured in a way that will feature both confidentiality and integrity. A protocol utilizing the cryptographic algorithm AES in CCM* mode will ensure that the aforementioned requirements are satisfied for the restricted nodes of the nSHIELD network.

The following table *summarizes the list of network requirements addressed by Prototype 24.*

Table 4-2: Prototype 24, Network Requirements addressed

Prototype 24: Network Layer Security prototype	
Confidentiality	Using AES in CCM* mode ensures the confidentiality of the communicated message.
Integrity and authenticity	The CCM* mode of operation utilises CBC-MAC as an integrity-checking mechanism. Correct reception of the message
Multiple protocol support	The scheme is able to support different cryptographic configurations, for increased compatibility.

4.5.3 Network layer prototype environment

The network layer prototype has been implemented in the Contiki OS and is meant for securing communication between nano nodes and other nSHIELD nodes. The implementation has taken place within the Contiki's uIP stack, which is responsible for handling the incoming/outgoing traffic of a node.

4.6 Metrics Approach (Prototype 27)

nSHIELD proposes 2 types of metric aggregation measurement. Both types of metrics are described in document D2.8 Final Metric Specification.: Attack Surface Metric and Multi Metric approach. Both need an integration procedure with respect the holistic nSHIELD platform.

This integration approach will be held by incorporating the aggregation formula to OSGI Middleware governing nSHIELD Overlay, and in particular by embedding it in the semantic model used in the SHIELD framework. This middleware will enable a container for aggregation formula and/or algorithm.

However it must be understood that both approaches have to be tuned by operator experts, so that integration will be finished with both perspective: this one which will be automatically addressed and one more manual one with the opinion of experts.

For railway scenario, many metrics have been found (See D7.1) and therefore, for multi-metric approach this could be seen as a good use case for its expert system algorithm validity.

4.7 Semantic model (Prototype 26)

The SHIELD Semantic Mode is based on a separation paradigm: on one hand a set of scenario dependent DBs contain all the information necessary to tailor the system aspects to the specific application. On the other hand, the abstract model of the generic SPD functionality, derived from the "attack surface" logic, contains the quantification of the SPD capabilities of the component as well as the mapping between menaces and means of mitigation (the basic principles of security). This concept is depicted in Figure 4-2 and detailed in D5.2 and D5.3.

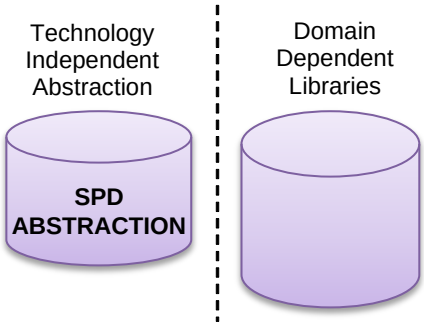

Figure 4-2: nSHIELD Knowledge Bases

All the requirements related to data integrity and management are addressed by an adequate data storage mechanism (relational DB rather than global variable in system memory) with basic security functionalities implemented by the software environment.

The integration of this prototype in the common platform is done in two ways:

i. By providing the components' responsible with the "guidelines" to write down the Ontology model for their component as well as the Domain Dependent Library

ii. By codifying this ontology into an xml file that can be parsed by the OSGI to extrapolate relevant information.

For the sake of simplicity, the demonstrator could be set up with the semantic data bases already initialized (in the real system this "learning" phase will be done at the "switch on")

4.8 OSGI Middleware (Prototype 25)

As already written in D7.1, D7.2 and D7.3 (from which this text is taken), it has been decided to adopt the open source Knopflerfish OSGi service platform to implement the behaviour of the SHIELD Middleware. Knopflerfish (hereafter referred as to KF) is a component-based framework for Java in which units of resources called bundles can be installed. Bundles can export services or run processes, and have their dependencies managed, such that a bundle can be expected to have its requirements managed by the container. Each bundle can also have its own internal classpath, so that it can serve as an independent unit, should that be desirable. All of this is standardized such that any valid Knopflerfish bundle can be installed in any valid OSGi container (Oscar, Equinox or any other).

Basically, running OSGi is very simple: one grabs one of the OSGi container implementations (Equinox, Felix, Knopflerfish, ProSyst, Oscar, etc.) and executes the container's boot process; much like one runs a Java EE server. Like Java EE, each container has a different startup environment and slightly different capabilities. The KF environment can be downloaded here: http://www.knopflerfish.org/

The KF start-up environment is shown below:

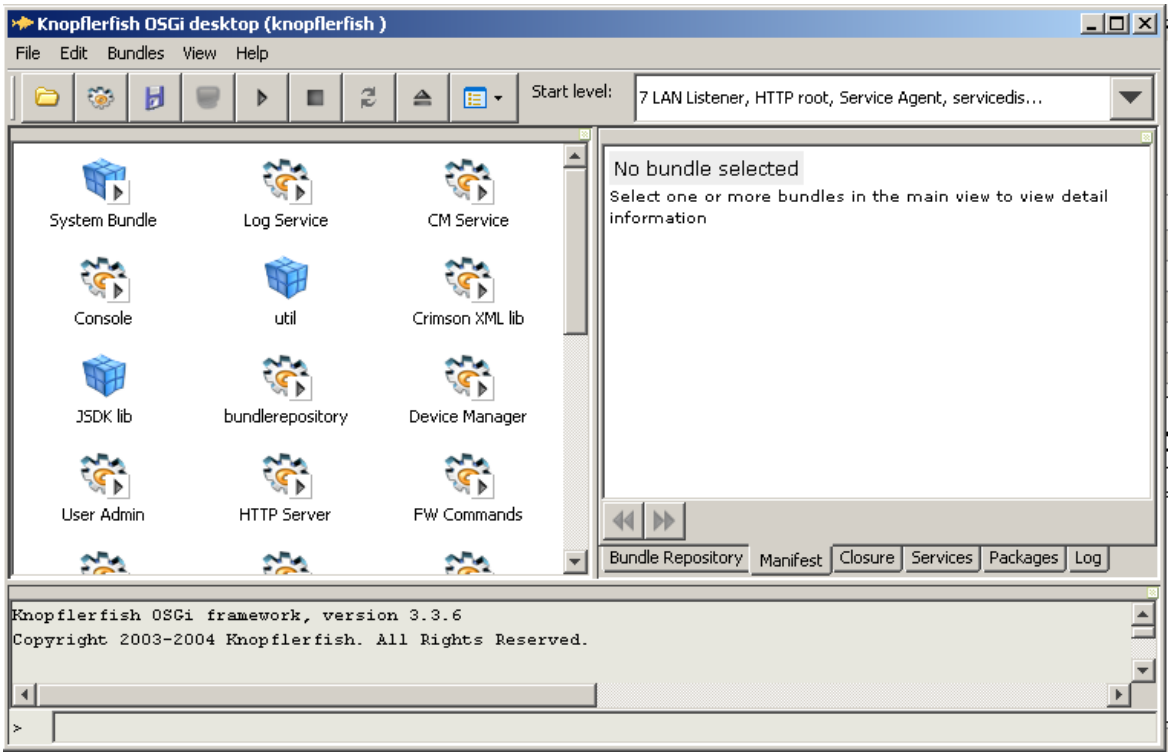

Figure 4-3: Knopflerfish start-up environment

All the major nSHIELD Middleware services have been translated into specific Bundles (including the Security Agent) so that the platform is representative enough of the final system.

On a deployment point of view, this framework is installed into a Notebook that is interfaced directly with the Intrusion Detection Bundle and consequently with the rest of the railways demonstrator through a network (most likely an Ethernet LAN). The interfaces with the Secure Discovery Bundle (in charge of populating the service databases) and the Security Agent are internal and implemented directly in Java Language.

4.9 Security Agent (Prototype 33)

The Security Agent is one of the main OSGi bundle and is responsible of interfacing the control algorithms with the discovery module, i.e. it represent the “embedded intelligence” of the SHIELD framework. It has direct access to the policy/ontology/domain repositories and can parse this information to feed the control algorithms. In addition, it is able to receive a solution computed by the control algorithms and translate it into a set of enforcement/control command to be propagated into the system directly or by means of proprietary protocols (in this case the security agent acts more like a remote console to drive specific equipment, i.e. the railways demonstrator PSIM server).

Since the Security Agent is an OSGI bundle, no integration issues are foreseen, since it is native in the middleware prototype itself.

4.10 Secure Discovery (Prototype 32)

The Secure Service Discovery protocol is in charge of discovering the SHIELD components (including their semantic description) and feeding the Security Agent with this information, useful for control purposes. A typical service discovery architecture could be depicted as follows:

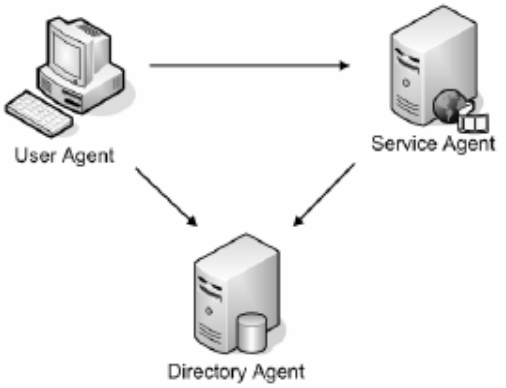

Figure 4-4: Service discovery architecture

Where the three entities present in all the demonstrator are depicted, and in particular:

- a Service Discovery Client (User Agent), which initializes the service discovery process: it is the entity interested in finding a certain service
- a Service (Service Agent), which, further being the service (one of the services) to be discovered by the client, is interested to be discovered, in case properly "advertising" itself
- a Service Repository (Directory Agent), which is a sort of database containing all the services that have published themselves in a certain scenarios and which discovery client could actually find.

The service discovery client and repository will be implemented in the OSGI framework, while the service agent will be installed into the railways demonstrator PSIM server. The specific discovery protocol adopted for the demonstration purposes is the SLP protocol.

4.11 Automatic Access Control (Prototype 11)

4.11.1 Automatic Access Control Prototype Interfaces

Access control mechanisms are in charge of preventing malicious entities to access the physical resources of a network node. Nodes utilize asymmetric cryptography to verify access requests. A DoS attack can be performed if a large number of access requests are sent to exhaust node's resources. Automatic access control embodies lightweight features to easy the verification process and avoid the DoS attack. The prototype is utilized in the railway scenario for providing automatic access control functionality between clients and a server, and preventing some types of DoS attacks.

In Figure 4-7, the general interfaces of interacting between a node and server are denoted as the prototype "Automatic Access Control – 11" under the "PSIM Server" component.

Gossamer [14] is a protocol for preventing DoS attacks on RFID systems. It belongs to the UMAP (Ultra Lightweight Mutual Authentication Protocol) family of protocols and provides data confidentiality, tag anonymity, mutual authentication, data integrity, forward security, robustness against replay attacks and DoS attack prevention. We implement the Gossamer protocol for the automatic access control prototype and Node Layer dependable self-x technologies. The protected server and its clients utilize the prototype prior to a session communication to achieve the aforementioned properties. Then, they continue the main communication tasks that are provided by the server. When the protocol fails to recognize a legitimate user, it may lead to the conclusion that the system is under attack. Thus, other mechanisms like anomaly detection, intrusion detection, intrusion prevention, intrusion tolerance and mitigation, intrusion response mechanisms and firewalls can undertake.

The nSHIELD Node functionality will be responsible to monitor SPD properties and control desired SPD level for the prototype. For more details, see next chapter. Further information and source code for the automatic access control prototype is available in D3.2 [13] and D3.3 [7].

4.11.2 Automatic Access Control Prototype SPD features

The preliminary version of the automatic access control Module provides the following features for the Node services:

- *Report of a DoS attack*
- *Provides function interface to query Service Metrics that can be used to assess SPD level of the prototype:*
    - *transactionRate*
    - *Blacklist/whitelist additions and removals*
    - *Failed authentication*

4.11.3 Automatic Access Control Prototype environment

The automatic access control prototype was designed to become part of the Node services. Its function interfaces are implemented in C++ and implements the ultra-lightweight protocol for automatic access control and mutual authentication – Gossamer. The code is tested under the operating system Linux. The core functionality is implemented in the Linux kernel and rest functionality in the user space. The compilation creates a module in the Linux framework Netfilter, which is responsible for manipulating the out/incoming network traffic of a node.

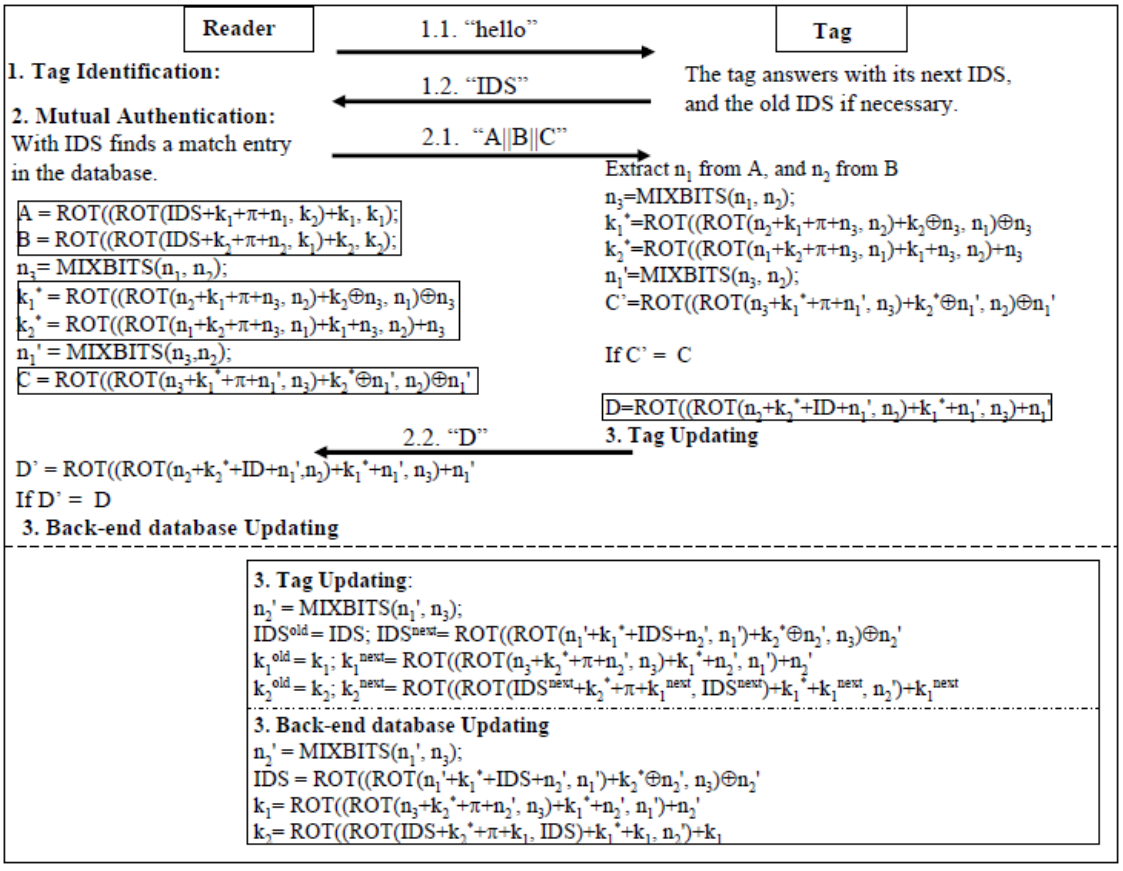

Figure 4-5: Gossamer protocol

4.12 Policy-based Access Control (PBAC) & Policy-based Management (PBM) (Prototype 19)

The nSHIELD secure policy-based access control (PBAC) framework facilitates the control of access to devices and their resources via security policies residing on resource-rich infrastructure nodes.

The PBAC framework is DPWS-compliant, utilizing the relevant specifications and existing work to provide message-level security and fine-grained security policy functionality while maintaining interoperability with the standard. The Devices Profile for Web Services (DPWS) is the "UPnP[1] for the Internet of Things"; a unified protocol platform developed because of the need to implement dynamic and secure discovery of devices and Web Services (including messaging, description, interactions, event-driven changes etc.) on resource constrained devices and supported by Microsoft and other industry leaders. While UPnP and DLNA (Digital Living Network Alliance) are favored for home entertainment scenarios, DPWS is recommended for enterprise and vertical applications. By adopting a DPWS-compliant mechanism, the PBAC framework offers seamless integration (discovery, access etc.) of new devices into the ecosystem and good scaling, which is especially desirable in large-scale deployments as will often be the case in Railway scenarios.

The solution adopted for secure policy-based access control is based on eXtensible Access control Markup Language (XACML) policies, an XML-based general-purpose access control policy language used for representing authorization and entitlement policies for managing access to resources and, moreover, an access control decision request/response language. The above fit well into the model of a network of heterogeneous embedded systems where access to resources is provided by nodes as a service, and into the management architecture developed by IETF Policy Framework. This typical policy based access control architecture combined with XACML is mapped to a Service Oriented Architecture (SOA) network of nodes to provide protected access to their distributed resources.

By combining the above technologies, the PBAC framework allows for fine-grained, policy-based control of all resources from remote locations, via any compatible app developed for the purpose or even typical browsers and off the shelf mobile phones. The resources may include but are not limited to DPWS-enabled cameras deployed on train stations or train wagons, sensors (to detect open emergency doors or environmental monitoring on carriages carrying sensitive material), control stations and other "smart devices" that are expected to be deployed in the context of a smart railway deployment. The framework will, therefore, facilitate access to various pertinent resources (e.g. sensor data or video stream), setting updates or even the receipt of alerts (e.g. in case of emergencies), all based on what the active policy dictates, while various metrics will be reported to the overlay.

The figure below depicts the integration of the PBAC framework into the Railway scenario System Security Architecture.

[1]Universal Plug and Play

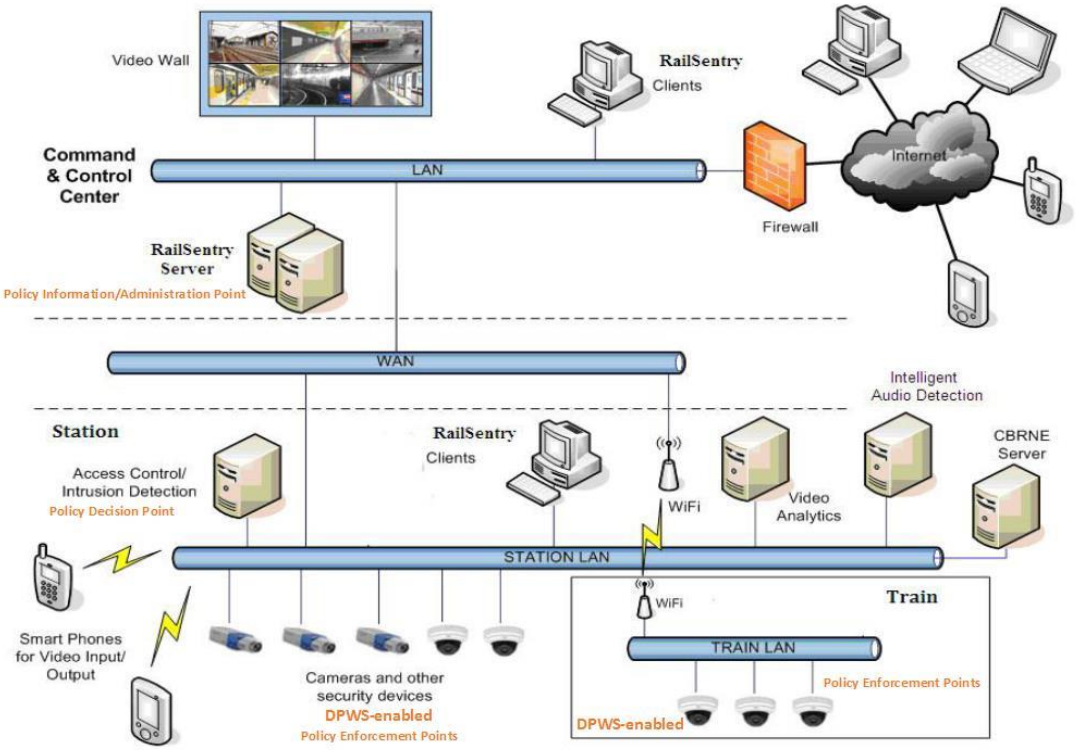

Figure 4-6: The nSHIELD secure policy-based access control

4.12.1 Policy Based Access Control SPD & integration features

The aforementioned framework addresses the critical issue of controlling access to nSHIELD resources. Its main features are that (1) it is policy based, hence allows the dynamic change of privileges and the SPD levels, based on the stakeholders' needs and decisions and (2) it provides the capability to directly access nodes' resources and address access requests to them, with no need to be aware of the system's details. In that sense policy based access control only authorized access satisfying the corresponding requirements.

There is a strong relation between the SPD levels of the PBAC framework and the defined policies and the corresponding rules which can be very strict or relaxed based on the system owner's requirements. Note that this policy can be defined either on a node basis, set of nodes, or for the whole system or it can even target specific subjects and resources.

On top of these policy-based levels there are some features available that affect the protection of the framework itself and therefore its effectiveness. These mechanisms are the encryption of messages and their authentication. Unprotected messages can disclose access control related messages and make them subject to unauthorised modifications. Therefore, the SPD levels of the Policy Based Access Control mechanism are shown in Table 4-3.

Table 4-3: PBAC SPD levels

SPD Level	Functionality
1 (low)	No encryption – No Authentication-
2 (medium)	Encryption or Authentication
4 (high)	Encryption + Authentication

The various entities (DPWS devices/peers) that the PBAC framework is comprised of can report various metrics to the higher layers, as needed. In terms of other integration features, an OSGi-DPWS interface has been selected and integrated into the standard OSGi platform typically running on nSHIELD power nodes (i.e. Knopflerfish). Said interface uses DPWS as communication protocol and includes several features regarding the mutual integration of DPWS and OSGi.

4.13 Interactions map

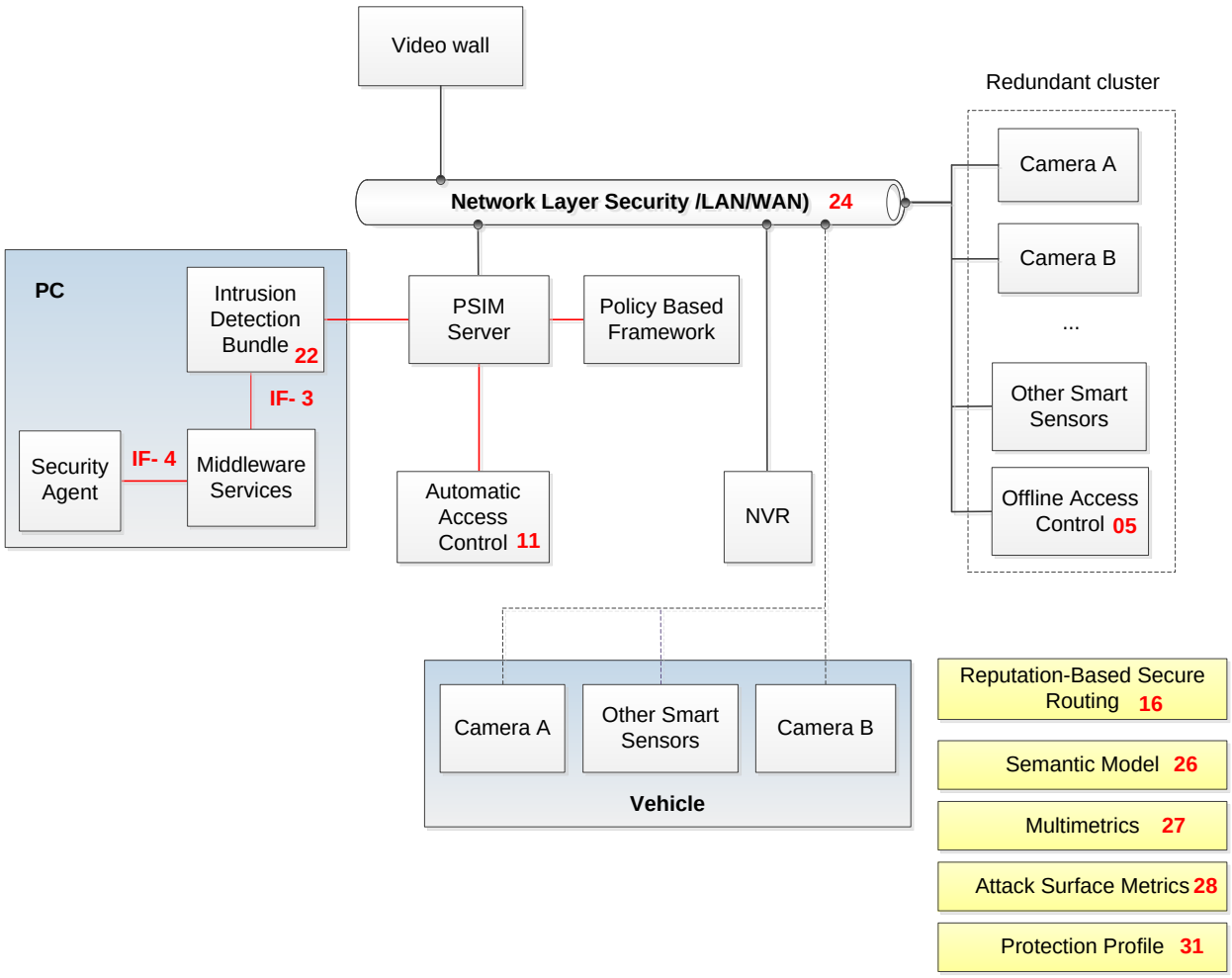

Figure 4-7: Railway scenario interactions

5 Integration of People Identification Scenario Components

5.1 Face recognition (Prototypes 7 and 37)

The prototypes 7 and 37 are the face recognition components of the scenario “People Identification at the Stadium”. This scenario is an evolution of the original scenario “Face and voice recognition” and it has been introduced to better illustrate the capabilities and potentialities of the adopted technologies and for its relevance in terms of market interest.

The face recognition prototype (prototype 7) is an all-in-one solution conceived to practically demonstrate the functionalities and potentialities of the face recognition system for people identification. It represents a proof of concept of the technologies adopted for the face recognition and it will be used to develop the final prototype. This prototype belong to the former “Face and voice recognition” application scenario and has been developed during the first part of nSHIELD project. During the second part of the project it will be finalized developing the embedded camera for face recognition that can be used in a real environment (prototype 37) and in the final demonstrator.

From a software point of view, the approach identified in the assessment phase has been implemented using three different modules that rationalize the recognition process. The prototypes, both the Windows and the Linux versions, are based on the Eigenface method. This method is based on the idea of extracting the basic features of the face: the objective is to reduce the problem to a lower dimension maintaining, at the same time, the level of dependability required for this application context. The core of this solution is the extraction of the principal components of the faces distribution, which is performed using the Principal Component Analysis (PCA) method. This method is also known in the pattern recognition context as Karhunen-Loève (KL) transform. The principal components of the faces are eigenvectors and can be computed from the covariance matrix of the face pictures set (faces to recognize). Every single eigenvector represents the feature set of the differences among the face picture set. The graphical representations of the eigenvectors are also similar to real faces and, for this reason, they are called eigenfaces. The PCA method is autonomous and therefore is particularly suggested for unsupervised and automatic face recognition systems. This software solution has been developed in C++ and has been compiled for Windows, (all-in-one demonstrator) and for Linux-ARM.(final embedded camera prototype).

The two prototypes integrate with the smart card security services, introduced by TUC, and with the dependable distributed computation framework, developed by UNIUD.

5.1.1 Face recognition modules

The face recognition and people identification application is based on three modules: the face finder module (FF), the ICAO module (ICAO) and the face recognition module (FR). These components cooperate to implement the recognition and identification procedure illustrated in the following figure:

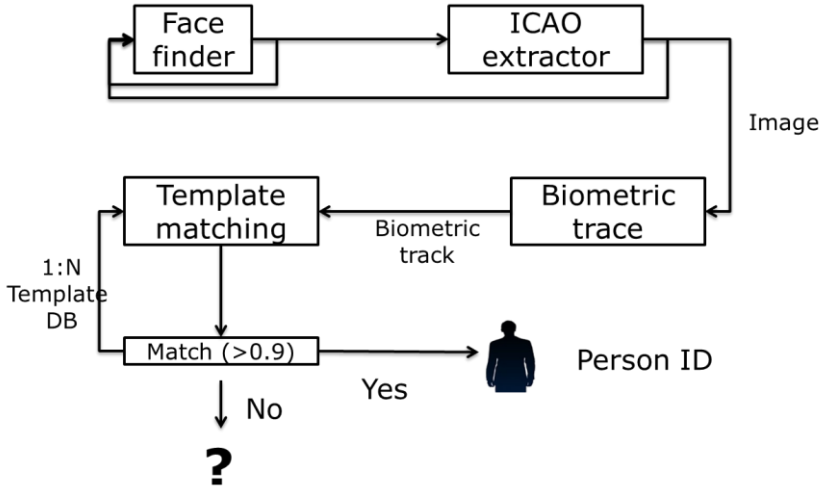

Figure 5-1: Face recognition and identification procedure

The face recognition application can operate in two different modes: “Enrol” mode and “Transit” mode.

The enrol mode is used to populate the data base with the biometric profiles of the people that will be accepted by the system. The transit mode is used to dynamically recognize and identify the people that pass (“transit”) in front of the camera. The following diagrams illustrate the operations performed in this two working modes:

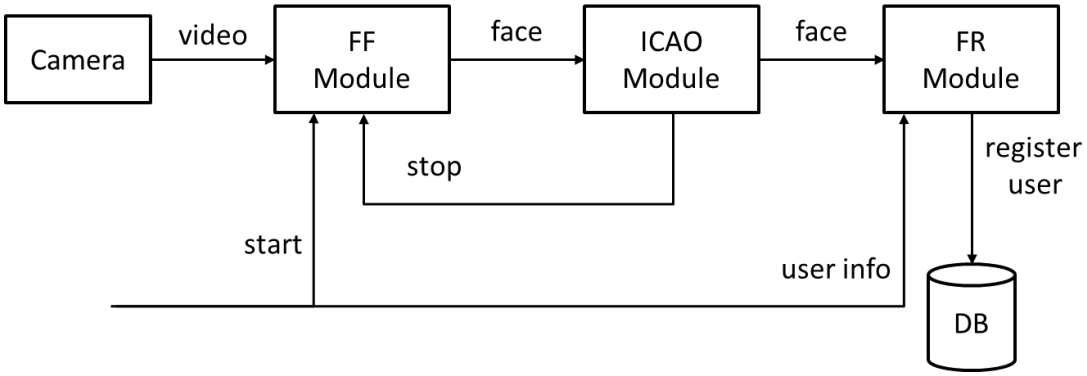

Figure 5-2: The enrol mode

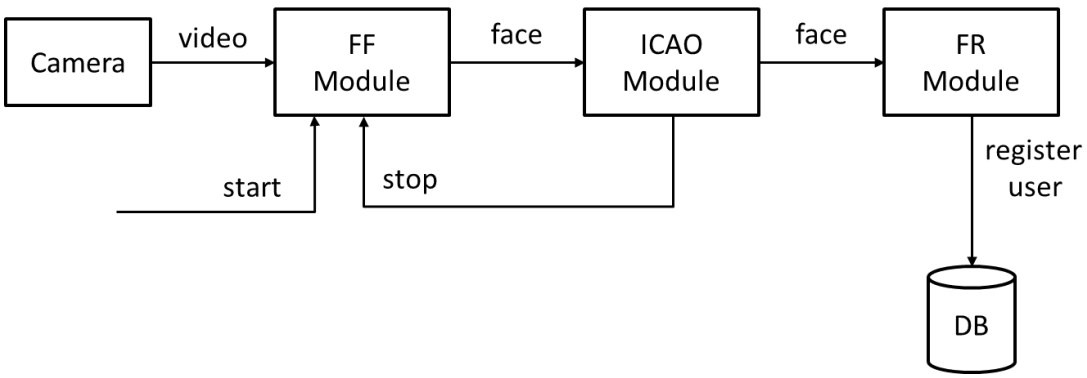

Figure 5-3: The transit mode

5.1.1.1 Face Finder Module (FF)

The face finder module is responsible for the acquisition of the video stream, for the analysis of the video stream itself and for the generation of the output messages when a human face is found in the input stream.

This module can be configured using a configuration file or through the web interface.

The module can be controlled using RPC.

The functions that can be executed by the application (RPC Server) are:

- Start: EthFF_Start()

  Start the process of face detection in the input stream.
- Stop: EthFF_Stop()

  Stop the process of face detection in the input stream.
- KeepAlive: EthFF_KeepAlive()

  Provide a feedback on the correct status of every module in the application.

Every time a face is detected in the video stream, the extracted features are passed to the ICAO module that accepts them as input data and elaborates them with an appropriate function.

5.1.1.2 The ICAO module (ICAO)

The ICAO module is responsible for the selection of the best image in the set of images identified by the FF module.

This module can be configured using a configuration file or through the web interface.

The module can be controlled using RPC.

The functions that can be executed by the application (RPC Server) are:

- Face: EthICAO_Face(DetectionData)

  This function requires in input the features of a detected face and provides the corresponding ICAO scores.
- KeepAlive: EthICAO_KeepAlive()

  Provide a feedback on the correct status of every module in the application.

Once the module identifies a face with an ICAO score that allows the identification, the “stop” function of the FF module is called and the best result of this detection phase is sent to the face recognition module.

5.1.1.3 The face recognition module (FR)

The face recognition module is responsible for the extraction of the biometric profile and for the identification of a matching profile in the data base. This module can run in two different modes: “Enrol” and “Transit”. In “Enrol” mode the detected biometric profile and personal information of a person are stored in the data base. In “Transit” mode the module search the data base for the biometric profile matching the one extracted from the video stream.

This module can be configured using a configuration file or through the web interface.

The module can be controlled using RPC.

The functions that can be executed by the application (RPC Server) are:

- Face: EthFR_Face(DetectionData)

  The module receives a face in input and extracts its biometric profile.

- User Info: EthFR_UserInfo(UserInfo)

  If the module is working in enrol mode, this function provides the user information of the person that has been detected and that will be registered in the data base. If the module is working in transit mode, this function provides the information of the identified person.

- KeepAlive: EthICAO_KeepAlive()

  Provide a feedback on the correct status of every module in the application.

In enrol mode the module performs the following actions:

- the module receives the image and the features of a face;
- the function EthFR_Face extracts the biometric profile of the face;
- the function EthFR_UserInfo provides the associated person information;
- the module saves the biometric profile and the person information in the database.

In transit mode the module performs the following actions:

- the module receives the image and the features of a face;
- the function EthFR_Face extracts the biometric profile of the face and compares it with the biometric profiles stored in the data base;
- the function EthFR_UserInfo provides the information of the identified person;
- the module saves the transit of the identified person.

5.1.2 Smart card manager

This component is responsible to read the encrypted biometric profile of a person from his/her smart card. It works with the access rights delegation module to setup a secure session during which the biometric profile is collected from the smart card using a common smart card reader.

This component provides the following functions:

- EthSD_OpenSession(SessionKey Key)

  Open a secure session using the session key generated by the access right delegation module.

- EthSD_GetHMAC()

  Obtain the HMAC generated by the access right delegation module.

- EthSD_GetBioPro()

  Read the encrypted biometric profile from the smart card during the secure session.

- EthSD_CloseSession(SessionKey)

  Close the secure session opened using the provided session key.

5.1.3 Smart card reader

The identification of a person is based on multiple sources of information, in order to increase the security of the recognition and identification process. The use of a smart card, that contains the biometric profile of the person, represents a solution to verify the identity of a person using two completely different sources of the biometric profile itself. Furthermore, this information is provided with different devices, with a very different physical nature and usage. This double check is performed at the turnstile when the person is in

front of the camera and the verification software asks to him/her to insert the smart card in the smart card reader.

The smart card reader that has been selected for the demonstrator is the SCR3310 V2 - USB Smart Card Reader. It supports every ISO 7816 Class A, B smart card. Any USB that can support this standard can be used instead of the selected one.

The manufacturer developed an SDK that provides demo, tools and PC/SC source code samples for development in VC++, Delphi, C#, VB.NET. The selected smart card reader is supported under Linux: drivers and development information can be found on the web site [16].

5.2 Dependable Distributed Computation Framework (Prototype 14)

The Dependable Distributed Computation Framework (officially named Atta, for short) is a middleware that allows applications to be run on multiple nodes in a distributed way. It essentially enhances dependability by managing nodes redundancy and also adds security both in the registration of the nodes and the transmission of data.

The role of Atta in the Face Recognition scenario is to add the cited features to the ETH SecuBoard prototype (prototype 37). In this context, the face recognition routines in prototype 37 are written as library objects that Atta is responsible to deploy, (possibly) compile and run on the nodes of the distributed platform.

Interfacing with Atta is mostly a design concern: an application must be (re)designed using a specific dataflow model. Each vertex of the application model contains code, while each edge represents a data transfer. The effort from the designer is therefore to think in terms of independent sections of code that interact with each other.

It must be noted that Atta does not force the designer to abandon his compilation and testing flows of choice: as soon as an application model has been designed, each code section can be built and tested in isolation using the preferred tools. With little adaptation, all sections can be linked together to test the whole application in a monolithic way. Consequently, while an “Atta-aware” design introduces additional concerns, it still accommodates as much as possible the existing design conventions.

5.2.1 At a glance

In practice, an Atta application model is a set of descriptor files in the YAML language. There may exist several descriptor files since the application is split into different so-called *artifacts*, one for each descriptor file, that combined with each other realize the complete application. This approach favours the reusability of code and also helps manage complex applications. The choice of the YAML language versus the XML language is for human readability, since YAML uses indentation rather than tag pairs as delimiters, allowing very terse and intuitive descriptors.

Figure 5-4 shows how a new application can be designed by using the Atta paradigm under a top-down approach. First, the application is defined as a directed graph. In this phase the semantics of each code section is undefined: only the interactions between code sections are important. Two kinds of artifacts are hereby identified: the data *types* for the information transferred between vertices and the blocks (called structural implementations, or simply *structures*) of vertices interconnected by edges. A vertex within a structure may contain either another structure or a code section. The artifacts corresponding to the latter are called behavioural implementations or *behaviours;* the specification of behaviour describes how to build the sources and access the resulting library. Please note that each artifact will have its own descriptor file and such file will be published (along with any source code, in the case of behaviours) in a versioned repository for deployment.

When all the artifact descriptor files are produced, the coding phase can start. This phase is conventional, in the sense that scripts or library objects for behaviours can be written with no particular concern for Atta-related aspects: it is only sufficient to actually supply the access function (similar to the “main” function of C/C++) declared within the descriptor file.

As soon as all the code artifacts are written and published, the work of the designer is complete. If the application has to be run in the distributed platform, the middleware will be responsible for downloading artifacts, generating and building the “glue code” that allows to actually interfacing the code sections together.

If instead the application already exists, the designer should identify the independent code sections and split his code base accordingly. Then, the descriptor files for the corresponding behaviours can be written, along with any types required for interfacing; finally the descriptor files for the structure(s) and the remaining types that connect the behaviours are provided. This represents a bottom-up approach that is still perfectly valid, especially when we expect the application to grow in complexity during the development phase.

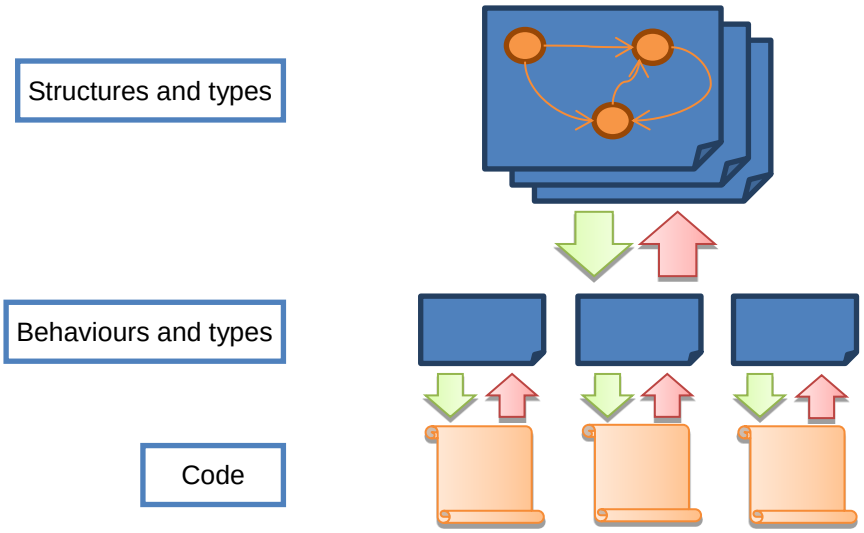

Figure 5-4: Top-down (green) and bottom-up (red) design flows in Atta

Summarizing, the interface that Atta exposes is essentially the set of artifact descriptors, where a certain artifact is a structure that represents the whole application. Since such structure “imports” the remaining artifacts, the application is ultimately represented by one artifact only.

In the following we will touch the most important aspects of the artifact descriptor files for types, behaviours and structures. As a prerequisite, we will describe how references to repositories are declared within a descriptor file. A complete coverage of the specification turns out to be overwhelming for the scope of this document: we rather refer the interested reader to the Latex documentation of the framework. The latest version can be obtained by checking out the public Git repository at the official Atta web site [12].

5.2.2 Repositories

Repositories are fundamental in order to both identify artifacts in an unambiguous and secure way, and to provide a reliable source for deployment.

A repository may contain multiple artifacts, each one identified by a descriptor file. For this reason, the declaration of repositories is separated from the declaration of references to artifacts.

```
repositories:
- name: myrepo
  control: git
  address: 'git@mysite.com:myself/myproject.git'
  mirrors:
  - 'git@thirdsite.com:myname/myproject.git'
  - 'git@fourthsite.com:othername/someproject.git'
```

```
references:
- name: vector
  repo: myrepo
  path: myproject/physics/vector
  version: 477125dbe78fe0a51be2486d8902b49ec2161450
```

Figure 5-5: Example of declaration for repositories and artifact references

In Figure 5-5 we can see how both repositories and references are declared.

For a repository, the *name* is the name given to the repository in the domain of this descriptor. The *control* represents the version control system used. We support the *git, svn* and *hg* systems, corresponding to Git, Subversion and Mercurial. The *address* represents the main coordinates for downloading artifacts. Finally, the optional *mirrors* represent equivalent implementations that are alive, i.e., exist simultaneously. We provide the ability to specify mirrors in order to avoid a single-point-of-failure situation where no node is able to collect a required artifact.

For a reference, *name* is the name given to the artifact in the domain of this descriptor; it can be used within the file to refer to the remote artifact. The *repo* references the declared repository in this descriptor, thus it must correspond to an existing repository name. The *path* provides the actual path to the artifact: following this path, an *artifact.yaml* file describing the artifact is expected to be found. Omitting the path implies that we refer to the root of the chosen repository. The version cannot be omitted, since assuming the latest commit is directly against having immutable references to artifacts. Please note how no branch information is used: the version and the path already provide all the necessary information to identify a specific state of the repository.

This information is therefore sufficient to refer to external artifacts, which can be downloaded from the declared repositories and combined into a self-contained application during run time.

5.2.3 Types

Since nodes may have heterogeneous CPU architectures, floating point precisions or running OSs, it is necessary to abstract type information away. Consequently, Atta uses a minimal set of *atomic* data types, namely *boolean, byte, real* and *text.* In practice, the real type covers all the numeric values, while the text type covers terminated char arrays.

Apart from atomic types, we allow *composite* types that combine existing types hierarchically.

For each of these types, multi-dimensional homogeneous arrays are supported, thus limiting to a regular (i.e., rectangular) matrix; hence we say that a data type is either a *scalar* (if no array specification exists) or an *array*. Arrays can be made of composite types freely.

```
kind: type
name: town
fields:
- name: name
  type: text
- name: location
  fields:
  - {name: province, type: text}
  - {name: region, type: text}
  - {name: state, type: text}
  - {name: coords, type: real, array: '2:3'}
- name: props
  type: text
  array: ':,2'
```

Figure 5-6: Example of artifact descriptor for a type

In Figure 5-6 we have the full artifact descriptor of a type. The *kind* field describes the artifact kind, along with its *name*; the name only has descriptive purposes, since references provide their own name for the artifact. The *fields* then supply a list of fields, possibly hierarchical, where each field has at least one *name* and a type. The *array* specification specifies that the field is of array type and its bounds. In particular, for *cords*, the array has one dimension with 2 to 3 elements. For *props*, two dimensions exist, separated by a comma, in particular an unbounded first dimension (because there is no value on both sides of the colon) and two elements on the second dimension (i.e., an N-by-2 matrix).

This is the basic way to declare types. We also allow a field to have a composite type as its type: it is sufficient to use the composite type name, and then include the corresponding reference and repository within the artifact descriptor. This approach supports type reuse and simplifies the type descriptors dramatically.

5.2.4 Behaviours

Behavioural implementations (behaviours, for short) are the code units of Atta. Given the dataflow paradigm of computation, they essentially represent (stateless) functions that take inputs and produce outputs.

A behaviour descriptor file has a *behaviour* artifact kind. It is characterized by the *language* element that specifies the language of the interface of the implementation. Currently, the *java*, *c* and *c++* values are supported.

Then there are several classes of specifications that define the implementation itself. We cover the most important ones, namely:

1. Entry: (optional) the entry point of the library/script used to start computation;
2. Ports: the interface of the implementation in respect to vertices that will be bound to it;
3. Build: the specification for building and linking the sources;
4. Load: the specification for loading the behaviour.

In addition to these specifications, we will have repositories/references and types, as described previously.

Before discussing these classes, let us introduce the concept of a *runner*, which is the Atta way of specifying some executable (binary or script) that has to be run in order to perform an action. A runner contains a list of elements called executable, each specific to an *OS* (*linux*, *macos*, *windows* and *solaris* are supported, but also **nix* to accept any among the non-windows variants). If no *os* element is absent, the executable is os-independent, meaning that it is run into an environment that is multi-platform (such as the Java or Python runtimes). Apart from the *os*, three other elements may be provided to an executable: a *command*, a *file* and an *args*. The command is a path-independent command that can be launched, like *make*, *ant* or *bash*. The file instead is an executable file (possibly with a path relative to the location of the artifact descriptor); we remind here that paths can be specified with either forward or backward slash, independently from the operating system. Either a command or a file can be specified, not both. Please note that a runner does not define the semantics of the underlying executable(s): it is the designer that must choose the proper runner for the required task.

Now the four specifications are summarily described. Please refer to the example of Figure 5-7 which covers the basic elements described below.

5.2.4.1 Entry

This is an optional specification that defines which entry point is provided by the behaviour; this specification is given within an *entry* element. Since different languages are supported (including scripts, in the future) it may be the case that the framework needs to identify how the entry function is accessed. Consequently, the entry element itself has up to four different elements under it, namely *file*, *namespace*, *class* and *function*. The *file* is useful for script languages and it defines which file contains the entry

function; the path is relative to the location of the artifact descriptor. The *namespace* provides the namespace, while the *class* adds the class name if pertinent (e.g., Java requires it, C++ may have a free function, C does not have classes). The *function* finally identifies the actual function. If a required element is not present, defaults are available: in particular, files are expected at the same location as the artifact descriptor, with name *atta* and extension dependent on the language. The namespace is empty if not available for the language, *atta* otherwise. The class is *Atta*, but only if required by the language. The entry function is always *atta_main*.

5.2.4.2 Ports

Ports describe the interface of the implementation in respect to vertices; this specification is given by a list within a *ports* element. Each port has some fields, where the most important are the *name* and the *direction*.

While the name is straightforward, the *direction* can have three values: *in*, *out* and *inout*. At least one input port and one output port (or alternatively, one in-out port) must be declared.

Also, the transmission of produced data can be made secure using encryption, by setting the *secure* field to *true*; this field is optional and it defaults to *false*. It applies only to out ports or in/out ports, the latter meaning that data is secured only in the output direction.

Please note that the order of declaration of the ports is also the order in which the arguments to the entry function have to be supplied.

5.2.4.3 Build

The build specification allows identification of how to build the sources of the implementation. All the specification lies within a *build* element. We currently support two kinds of build systems: *maven* and *custom*. The Maven build has several advantages, one of them related to the ability to discover dependencies; it is assumed that a *pom.xml* file is present in the root of the checkout path: this information is sufficient, in the default case, to describe the build. As for the custom build, we can provide actions for three separate phases: *prepare*, *clean* and *compile*. The preparation is done once and creates supporting files, like it happens with Autotools using the *autoreconf -i* command; if no preparation phase is defined, it is simply not performed. The build phase is the main phase (for Autotools, *configure* followed by *make all*) and its textual output can be analysed to identify errors during compilation. After fixing the issues, the build phase can be performed again, possibly preceded by the clean phase if present. No phase is really mandatory here: if no prepare/clean command is defined, its phase is simply ignored; if no phase at all is defined, then the script is simply executed as-is.

5.2.4.4 Load

Even when compilation and linking succeeds, loading the process that handles the behavioural implementation still may require some attention. This is particularly true when dynamic libraries must be loaded. For that reason, the Load specification provides specific runners that may be used for bringing up the process while satisfying all its runtime dependencies.

```
kind: behavior
language: java
entry:
  namespace: my.domain.examples
  class: AttaAccess
  function: compute
ports:
- name: ip1
  direction: in
  type: mytype1
- name: op1
  direction: out
```

```
  type: mytype3
  secure: true
- name: iop1
  direction: inout
  type: mytype1
build:
  kind: custom
  target: Build
  phases:
  - kind: prepare
    args: prep
  - kind: clean
    args: cl
  - kind: compile
    args: build
  runners:
  - os: *nix
    command: bash
    args: atrun.bsh
load:
- os: *nix
  command: bash
  args: run.bsh
- os: windows
  file: run.bat
```

Figure 5-7: Example of artifact descriptor for behaviour

5.2.5 Structures

Structural implementations (structures, for short) are defined in terms of vertices and edges.

It must be noted that a system model is just a structural implementation, possibly with inputs and outputs. Each structure can then be made of other structures, until behavioural implementations are found that actually supply the functions that are run in a distributed way.

Input and output ports may be specified, along with their types; in fact, ports are optional in some cases, i.e., where the structure is the whole system to be executed, since inputs are embedded in the model and outputs are only used internally. It must be remarked that the input ports and output ports are declared using a dedicated name, different from the edge names that will be wired to them. This solution allows to discriminate between the internal and external role of one data connection. If output ports are provided, each one must specify a wiring to an existing vertex or input (sub)port (more about wiring in the following). Compared to a behavioural implementation, no index or in-out ports are allowed.

In Figure 5-8 we show an example of a structural implementation descriptor; again, types, repositories and references are omitted for compactness. The list of vertices and edges are described next

```
kind: structure
local: true
ports:
- name: ip1
  type: mytype1
  direction: in
- name: ip2
  type: mytype2
  direction: in
- name: op1
```

```
  type: mytype3
  direction: out
vertices:
# ...
edges:
# ...
```

Figure 5-8: Skeleton of artifact descriptor for a structure

5.2.5.1 Vertices

For each vertex we define the interface and its implementations. It is usually the case that only one implementation exists. While we do not discuss multiple implementations here, these are supported in the form of a *switched system* with defined transitions.

Vertices support *guards*, i.e., conditions on the values of their input ports that allow the execution of the bound implementation; guards are useful for conditional execution. A guard consists of a disjunction of conjunctions of expressions related to port fields (or edge fields, as we will see in the following). In Figure 5-9 the full syntax of a vertex is shown, where a guard is present. All the entries of the list inside a *clauses* element are combined with an AND operation; the resulting predicates are then combined with an OR operation, meaning that the guard corresponds to: (ip1.u=2 AND ip2>=0) OR (ip1.v>20). For fields with the *real* atomic type, we allow the =,!=,<,<=,>,>= operators, while for the *text* atomic type only the equality and inequality operators are available.

An implementation must be associated with a vertex using a *binding*. It is necessary since the interface of an implementation may have some mismatch with the interface of the vertex.

These are the rules for binding:

1. All vertex output ports must be completely bound, exactly once;
2. All implementation input ports implementation ports must be completely bound, exactly once;

These rules guarantee that vertex outputs always can produce data, while implementation inputs always can consume data; also, it prohibits binding the destination more than once. Please note that for composite types, we can bind a subset of the source to the destination.

The *kind* of a binding refers to the side of the implementation, where the *input* side binds vertex input ports with implementation input ports, and the *output* side binds implementation output ports with vertex output ports, in this specific order. The separation between the two sides is for clarity. Then, the *from* field specifies the source, and the *to* specifies the destination.

```
name: v1
ports:
- name: ip1
  type: mytype1
  direction: in
- name: ip2
  type: mytype2
  direction: in
- name: op1
  type: mytype3
  direction: out
guard:
- clauses:
  - field: ip1.u
    expression: '=2'
  - field: ip2
    expression: '>=0'
```

```
- clauses:
  - field: ip1.v
    expression: '>20'
implementations:
- name: impl1
  bindings:
  - kind: input
    from: ip1.u
    to: X
  - kind: output
    from: Y.a
    to: op2
```

Figure 5-9: Example of declaration of a vertex

5.2.5.2 Edges

An edge is simply the connection between structure and vertex interfaces.

It can be enriched by a *name*, but that has only descriptive purposes. More important, an edge has a *from* field which represents a source subport and a *to* field which represents a destination port. The source subports may come either from the input ports of the structure, or from the output ports of the internal vertices. The destination ports may come either from the output ports of the structure, or from the input ports of the internal vertices.

The syntax for wiring a source/destination vertex is *v@p*, where *v* is the vertex identifier and *p* is a port or subport, using the dotted notation (e.g. *myinfo@position.x*). The syntax for the enclosing structure source/destination simply omits the vertex id, i.e., *@p*; the empty name avoids the introduction of a protected keyword, such as ``this'', to refer to the behavioural implementation.

In addition, an edge can have a guard too: if the guard is not satisfied, new data flowing through the edge is ignored. In this case, the guard is expressed in terms of the ports in the source of the edge.

An example of an edge is shown in Figure 5-10.

```
edges:
- from: @ip1
  to: v2@x
- name: speed
  from: v1@x
  to: @op3
- from: v2@y
  to: v3@p1
- from: v3@y
  to: v3@p2
  guard:
  - clauses:
    - field: p1.b
      expression: '=1'
```

Figure 5-10: Example of declaration of an edge

5.3 Smart Card Security Services (Prototype 6)

A smartcard is a tamperproof secure device resilient to physical attacks used to perform secure transactions. Smartcards are used in a plethora of applications require security such as payment applications, healthcare, physical access control to mention a few. Smartcards can provide multiple security levels for sensitive data stored in them. For instance, a security key can be marked as read-only,

while the read operation is accomplished only inside the smartcard. Even more the security key can be protected by a PIN to add one more security level. One of the main advantages of smart card solution is that all the sensitive operations are accomplished in the smart card rather than the terminal or application, which in many cases is not considered trustworthy. Smartcards among to others provide the following security services:

1. Message Authentication code
2. Encryption
3. Identity validity
4. Digital signatures
5. Hash functions
6. Secure key management

5.3.1 Communication with Smartcards

Smartcards have the structure depicted in the figure below.

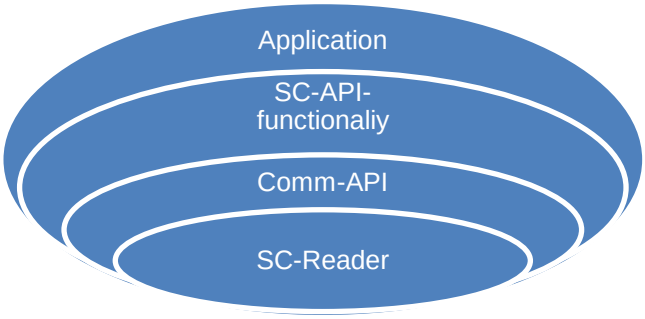

Figure 5-11: Smartcard communication structure

It should be noted that even in cases that smartcards do not provide a specific API for communication between the application and the smart card the communication with them can be accomplished by issuing direct command to the smartcard since the smartcards follows the ISO standards [5]. The general structure of a command in smartcards is illustrated in the table below.

Table 5-1: Smartcard request command format

Header				**Data**	
CLA	***INS***	***P1***	***P2***	***Length***	
Class where the command lies	*The command itself*	*Command first parameter*	*Command second parameter*	Data Length	Additional Data

The command can be issued towards the smartcard using the underlying communication of the terminal and the smartcard terminal (e.g. serial communication).

For every command issued toward to the smartcard there is a response which its format illustrated in the following table.

Table 5-2: Smart card response command format

Data	*Response Status*
The data returned by the smartcard	*Show the result of the requested command ,whether the command is successful or failed, and the reason of failure*

5.3.2 Smartcard File System and Data "Storage"

Smartcards file system structure is similar to those used in operating system. Particularly the ISO-7816 part 4 defines the structure of the file system as illustrated in the following figure. The master file (MF) can be considered as the root directory, while the dedicated and elementary files are the directories and the data file, in UNIX like operating system, correspondingly.

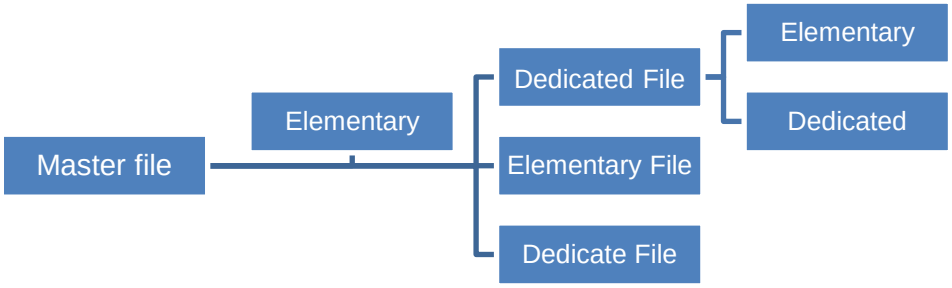

Figure 5-12: The logical structure of file system in Smartcards

- In smartcards different kind of data can be stored either dynamically or statically, though their capacity is limited. For example, users' data or cryptographic keys for secure transactions can be stored. The header in data files defines also the access control rights. Every directory creates a security domain inheriting the security policy of its parent. The files in the smartcard can be protected with multiple ways:
- Different PIN
- Message authentication code
- Access control restrictions (read, write permissions)
- Digital signatures
- This depends on the features incorporated in the smartcard.

5.3.3 Secure services with smart cards

Depending on the type and the manufacturer the smartcards support a number of cryptographic features, including:

- On-card generation of symmetric keys and public key algorithms key pairs
- Digital signatures (based on public key algorithms)
- Symmetric encryption and decryption
- External authentication (host to card)
- Internal authentication (card to host)
- Message authentication code
- Hash functions

Further, smartcards enable protected mode for highly sensitive data, which requires commands to be authenticated and integrity protected either with symmetric or asymmetric keys.

5.3.4 Building Secure Communications

In order to build trust among different type of nodes on the nSHIELD architecture we can exploit the benefits of smart cards and the cryptographic schemes they implement. Considering, the nSHIELD architecture where decentralized components are interacting not only with each other but also with centralized ones, depending on the type of the device and the employed scenario; there is a need for integrating security and interoperability. In this context, we rely on [14] for building secure communication channels among different devices. We should mention that smart cards currently are used in various applications, where proof-tamper devices are need for the provision of security services.

In the proposed scheme in order to issue a smart card the related component (e.g. micro node) should create a request for issuing a smart card. This request will include the serial number of the component and will be forwarded to the central authority. If needed, depending on the type of service, the central authority will check the register status of the requested component and afterwards will generate a new smart card. In the new smart card will be installed the following information:

- Node's serial number.
- Node's secret key.
- Node's id.
- Node's auth key.

The generation of secret keys will be based on the following types:

```
Encryption-Key = AES-256 (Central Mother Key XOR Node's Serial Number)
Auth-Key = AES-256 (Central Mother Key XOR Node's ID)
```

We should note that in this scheme we assume that the central authority has also a TPM for generating the secret keys in a secure way for the issued tokens, while all the smart cards are issued by a (trusted) central authority. The generated keys will be unique since they are related with node's serial number, which is unique.

The node, as a smart card is issued can exploit its security feature for providing confidentiality, integrity or/and authenticity services. The provided services depend on the application. For instance, if there is a requirement to provide confidentiality services to the data sent to the central authority the following procedure will be take place:

- The node will send the data to the smart card.
- The smart card encrypts the provided data, using the secret-key installed into the smart card during the registration and forwards them to the node.
- The node sends to the central authority the encrypted data and its serial number.
- The central authority generates in the TPM the corresponding secret key using the serial number sent by the node. Note that the key is not "extracted" from the TPM.
- The TPM decrypts the data and send and acknowledgement to the node.

A very similar approach will be followed when an authentication is needed. Particularly:

1. The node will send to the smart card a random number that will be used as the data require validation.

2. The smart card using the Auth-Key and the random data generates the MAC and forwards it to the node.
3. The node sends to the central authority the MAC including the random number and its id.
4. The central authority generates in the TPM the corresponding auth key using the id.
5. The central authority using the auth-key produces a new MAC and compares it with the one received by the node. If those two MACs are matched the central authority sends to the node a successful response otherwise a failure occurs.

These procedures can be combined in order to provide confidentiality and authenticity services simultaneously, depending on the requirements.

5.4 Access Rights Delegation

Within the scope of 3.2 (WP3, Task 2) an approach to delegation of access rights has been investigated. In a network of offline trusted embedded systems, a node need to be able to authenticate another node requesting some privileges, but also to determine what – if any – privileges should be granted. A model for doing this has previously been developed by the project partner (Telcred), but this model assumes that all access rights are issued by a central trusted authority and does not support delegation.

This solution can be adopted in the "People Identification at the stadium" scenario, both for the identification of people at the turnstile and for identification and access granting to the personnel of the stadium.

5.4.1 Problem Statement

In an offline PACS (Physical Access Control System), there is no continuous exchange of information to verify and allow a user through a series of doors, whereas this is a common feature in an online PACS. Current offline systems are unable to force a user to follow a certain designated route, e.g. Room A should be accessed before entering room B. This project explores a model to enforce such a route, by using delegation of some authority from the main administrative system to the offline locks.

5.4.2 The Concept of "Path Array"

The developed artefact consists of a construct known as Path Array aka PA. Path Array is an array that can be one or multi-dimensional based upon the administrator requirements. PA consists of $\texttt{Lock}_{id}$ stored into each index of the array that needs to be accessed by the user in a sequence. Administrator is responsible to implement path array onto the user's smart card before handing it over to the user.

`Ticket` that is stored in the flash memory of the smart card contains the PA. After the formation of mutual trust between the `Lock` and `Card`, `Lock` makes use of the remaining contents inside the `Ticket` for decision making.

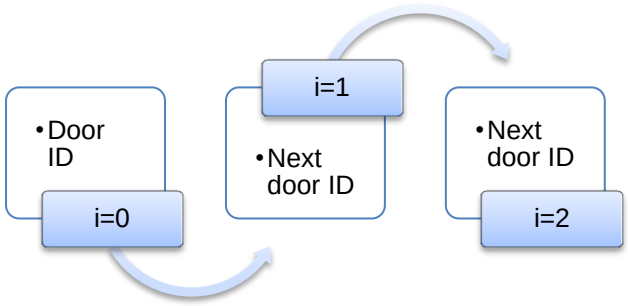

Figure 5-13: Path Array Design

The figure above shows the outline of PA. PA consists of $Lock_{id}$ stored at each index (i = 0, 1, 2...). PA holds the $Lock_{id}$ that should be accessible by a user in a sequence. `Server` digitally signs the PA stored inside the `Ticket`. Index i value starts from 0 and increments each time a user passes a door or a turnstile. This index value points to the $Lock_{id}$ that the user needs to visit. Hence, the contents of the `Ticket` will be as follows.

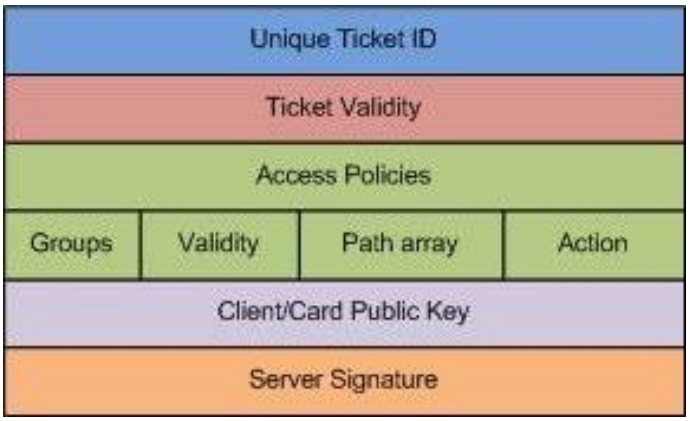

Figure 5-14: Ticket along with Path Array

5.4.3 Mechanism of the Artefact

After the creation of trust between the entities of offline PACS, `Lock` now processes the contents of PA, and then checks for its own ID at the current index i, if it is found then `Lock` performs three steps as follows,

- Increment index i
- Generate HMAC and write it to `Card`
- Grant access to the user

If the $Lock_{id}$ present at index i does not correspond to `Lock` own id, it then it denies the access and logs the user action.

5.4.3.1 Incrementing Index i

The path array PA contains lock ids stored inside it. Only the relative matching `Lock` is allowed to increment i value by one. At the time of generation of `Ticket` by the `Server`, it also generates a HMAC to be used by the first lock in the PA. The `Lock` located at the first index of PA makes use of this HMAC to ensure that no illegal modifications are done on the smart card. The Index of PA starts from the value 0. For instance, consider the below path array. This path array consists of lock ids B, A and C which should be followed in that order by the user.

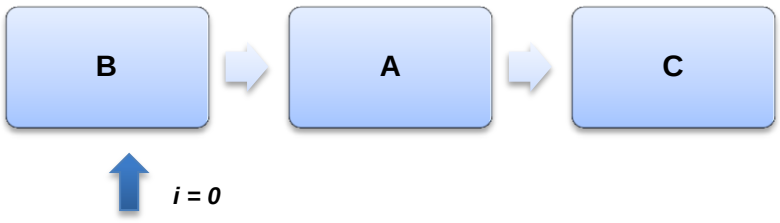

Figure 5-15: Ticket Incrementing the index value

In the figure above, the current value is 0 and PA[0]=B. Only the lock with id 'B' can increment the i value further.

5.4.3.2 Generating Hash using HMAC

Lock creates the HMAC after incrementing the i value. HMAC stands for Hash Based Message Authentic Code. It calculates the message authentic code using a cryptographic hash function and shared secret key. In the offline PACS scenario, geographically dispersed locks securely exchange the messages among them by using message digest. HMAC is necessary in offline PACS scenario to ensure the integrity of smart card contents. The process of creating HMAC is as shown in the formula below.

$$HMAC(K,m) = H((K \oplus opad) \parallel H(K \oplus ipad) \parallel m)$$

where:

- K is the shared secret key
- m is the message to be protected
- opad is outer padding (0x5c5c….)
- ipad is inner padding (0x3636….)
- H is the cryptographic hash function (MD5, SHA etc.)
- || is concatenation
- ⊕ is the exclusive-OR operation

The locks inside the facility were pre-installed with $\texttt{key}_{\text{shared}}$. Concatenating the $\texttt{key}_{\text{shared}}$ with $\texttt{Lock}_{\text{id}}$ generates the secret key $\texttt{key}_{\text{secret}}$. Message `m` in this context indicates the current index i value, and the rest of them use default parameters.

While hash generation, $\texttt{key}_{\text{secret}} = \texttt{key}_{\text{shared}} \,||\, \texttt{Lock}_{\text{id}}$.

Generation of HMAC is as following:

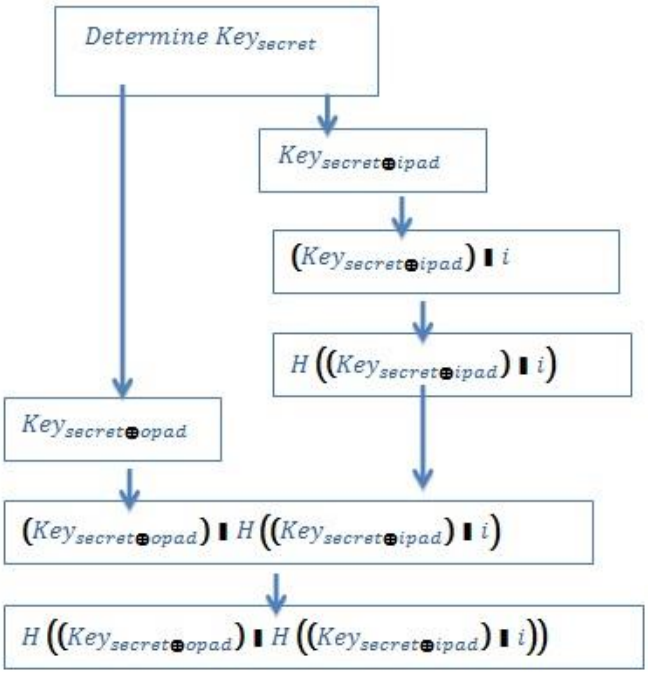

Figure 5-16: Process of HMAC creation

In the above figure, H can be any hash algorithm.

5.4.3.3 Generating Hash by the Locks in the Sequence

The overall concept of the artefact is to enforce the user through the path specified by the administrator. The user should attend this path in sequence. Hence, if one lock grants access to the user, which is present at index zero of PA i.e., PA[0], then the next lock in the sequence, which is available at PA[1] should be able to prove that the user has already passed through the door mentioned at PA[0].Verification of the hash value generated by the lock present at PA[0] solves the above issue.

When the user presents his smart card at a Lock mentioned in the PA, the lock allows access only if it has confirmed that the user has already been allowed access by an earlier lock. During the verification process of HMAC Lock always uses the previous index for key generation and while in a hash generation process it uses its own $Lock_{id}$. Current lock initially checks i value.

Scenario 1: If i=0

Then the lock knows that it is the first lock in the order. It then checks whether the value present at PA[0] matches its own lock id. If the id is not equal to its own id, it will log the user activity and deny the access.

Lock has the authority to perform further actions if the value at PA[0] matches its own id. It then verifies the HMAC stored by the Server on the Card, to make sure that nothing has been changed illegally. It will then increment i value by one and will generate HMAC by using the secret key key_{secret}. In this scenario, key_{secret} results from concatenating the key_{shared} with its own $Lock_{id}$.

Scenario 2: If i>0

If i value is greater than zero, then lock confirms that the user has already accessed some doors in the sequence. Hence, it will confirm its own authority that it can change the contents of the card by looking up for its own lock id, and then generates hash to verify the hash stored by the earlier lock. Now, the Lock increments i value by one and generate a new hash to be used by the next lock in the series.

Verification steps by current lock in action are as follows,

- Step 1: reads current i value
- Step 2: Looks up present at PA[i]
- Step 3: If the value of own =PA[i], then proceed to step 4 else go to step 10
- Step 4: Verify the HMAC hash stored on smart card (generated by previous lock)
- Step 5: If the hash can be verified, continue else go to step 10
- Step 6: Increment i value by one
- Step 7: Generate new HMAC
- Step 8: Replace the old HMAC with generated HMAC to be used by next lock
- Step 9: Allow access and stop
- Step 10: Deny access and log user activity onto the card

Using above procedure the n^{th} lock will verify that the user has accessed the $(n-1)^{th}$ lock, and this process continues with all the locks.

5.4.4 Smart Card and biometric data

Smart cards, among the others, can be used to store very sensitive data as those of biometric data (e.g images) in that way in which only authorized entities are entitled to get access to them. Biometric data provide high confidence with regard to the user identification and authentication because of their in

heritage properties. This means that the complexity to reproduce biometric data that belong to a specific entity is very high. Storing biometric data in smart cards can be used in order to achieve two factors authentication. This is because the biometric data can be stored securely in smart card in such a way that only the holder of the card can gain access to the biometric data.

5.4.5 Face Recognition Smart Card Support

To build trust among different types of nodes on the nSHIELD architecture we can exploit the benefits of smart cards and the cryptographic schemes they implement. In the nSHIELD architecture where decentralized components are interacting not only with each other, but also with centralized ones, there is a need for integrating security and interoperability. In this context, we exploit the advantages of smart cards to enable different types of nodes to provide the following security services:

- Allow the secure key management required for establishing secure channels between different nodes.
- “Anonymous” Authentication e.g., between the sensor and central or other distributed components in the train network.
- Protecting message integrity, for sensor data in the train network among the node and the central system.

In this context, smart cards as mentioned in D3.2 can be used in order to implement an off-line access rights delegation relying on Physical Access Control System. In this scenario the smart card will be used to:

- Generate the session key
- Generate the HMAC
- Store the Path Access

Furthermore, in order to increase the confidence which the service has to provide to the users about their claims regarding their identities, biometric data such as users images can be stored to the smart card and validated every time users trying to access a protected resource. For example, when a user tries to access specific doors the nSHIELD node captures an image of the user and compare its biometric data with those are stored in the smart card. This way, there is not a need to communicate with the central directory for all the requests. To incorporate this approach to the nSHIELD architecture the following figure depicts a high level approach for integrating smart cards security services in the scenario of face recognition.

5.5 Interactions map

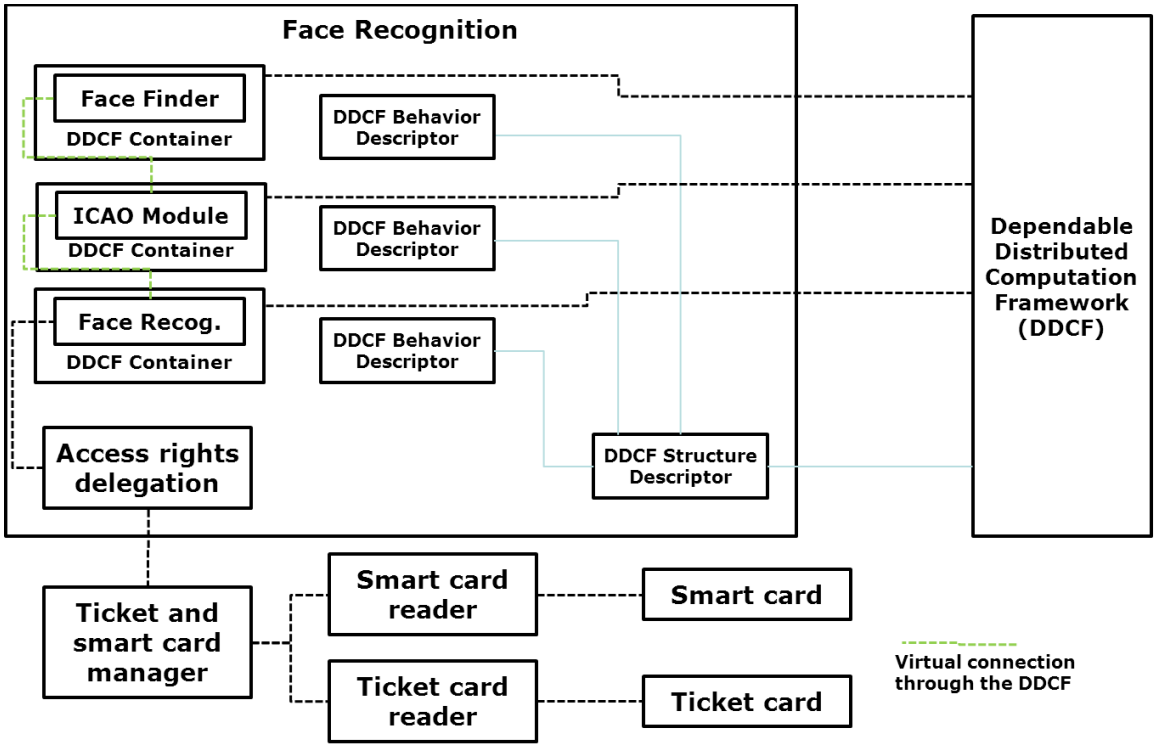

Figure 5-17: People Identification scenario interactions map

6 Integration of Avionics Scenario Components

6.1 OMNIA (Prototype 36)

OMNIA (Open Mission Network Integrated Architecture) is based on IMA and IMA2G concepts and introduces, at some level, typical nSHIELD dependability aspects, such as interoperability, fault detection, fault management, health monitoring and data integrity.

The idea behind OMNIA is to create an IMA platform composed by a network of several «Computer Units»; these can be HW boards or computers, acting as IMA CU (Central Unit), RIU (Remote Interface Unit) or both. Each unit acting as RIU is connected to the A/C (AirCraft) sensors. All units are “nodes” of a network, being connected by means of a High Speed deterministic serial line.

The OMNIA platform introduces the Middleware software to provide platform level services. The Middleware is implemented on top of the Operating System local to the hardware components. Its purpose is to enable IMA2G typical interoperability, allowing the provision of the same service with the same behaviour in such a way that it is independent from the physical location of the requesting applications, i.e. the hardware component that hosts the application, independent from the hardware component type, if applicable, and the Operating System hosted and independent from the location of the requested hardware resources

Platform Services are classified at two levels according to their scope, which can be the overall platform or a single hardware component; Platform Services include in fact Module Services. Platform Services are also classified according to their privilege, whether they support Avionic applications or Platform Management applications (including module management).

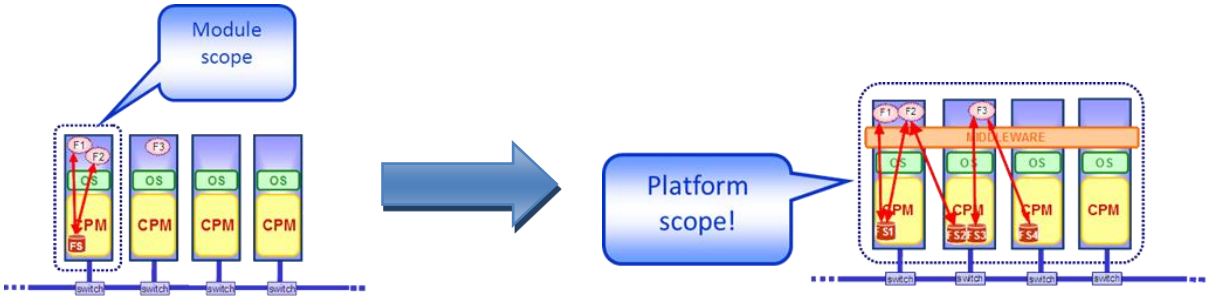

Figure 6-1: OMNIA Platform Services

More specifically, the “nodes” selected to create the OMNIA network are:

- The NAMMC (New Aircraft & Mission Management Computers)
- The APM460 processor module (stand alone)
- The NSIU (New Sensor Interface Unit)
- More types could be added in future

The NAMMC, including HW and Equipment SW, is a SES product, flying on board several types of aircrafts, able to host the customer Flight Management applications. It is currently completing the certification process. It mounts APM460 processor modules and is able to interface with the A/C sensors by means of IO boards (e.g. the DASIO). In the OMNIA platform the NAMMC can act as CU or RIU or both.

The APM460 stand alone can act as CU

The NSIU is a SES computer currently at development stage. It can act as CU or RIU in the OMNIA platform

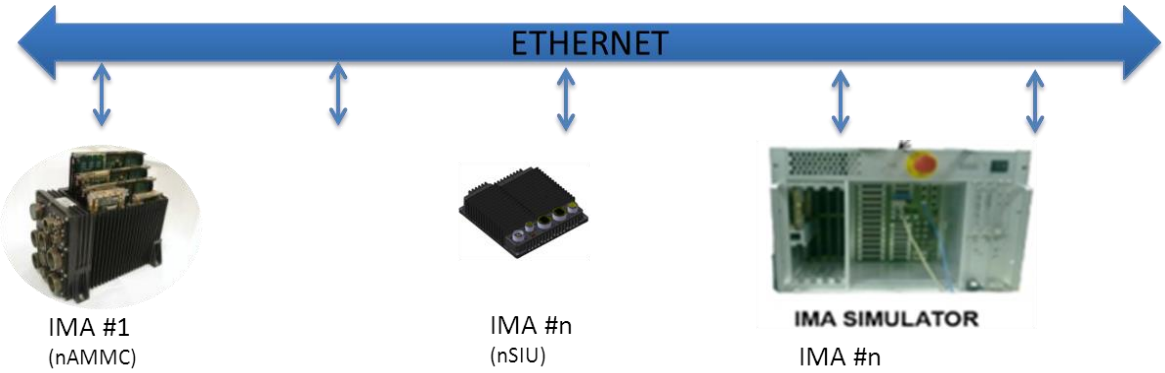

Figure 6-2: OMNIA network

For the nSHIELD avionic dependable demonstrator, as depicted in the previous figure, it is proposed to configure the OMNIA platform with two or more CU/RIU equipment (NAMMC or NSIU or simulated NAMMC/NSIU) in order to simulate the on-board avionics on a UAV. This configuration can be changed to include more “nodes” if necessary.

A SW middleware based on DDS architecture, with a unique service bus, will be used to “virtualise” the physical connection of the A/C sensors enabling **interoperability** within the OMNIA platform as it will allow the OMNIA platform “nodes” to access the sensor resources independently from the actual physical connection.

Health monitoring and **fault management** within the OMNIA platform are performed at “node” level by means of continuous built in tests. **Integrity** of sensors data will be handled at OMNIA middleware level

For the nSHIELD demonstrator the OMNIA system will be able to provide the main aircraft mission/navigation functionalities with all the relevant check either relating to the data integrity exchanged and to the integrity of the OMNIA system (e.g. fail, reconfiguration).

6.2 Gateway (Prototype 21)

nS-ESD-GW is a SHIELD framework pivotal component. This component has been introduced to foster the interconnection of nodes and to ease the SHIELD employment. Thus, in the context of the Dependable Avionic application scenario, this component will be employed to ease the integration and interconnection of the OMNIA framework to nSHIELD nodes, middleware and overlay as well. The nS-ESD-GW will exploit the flexibility of the SoC (Zynq) to provide appropriate interfaces towards the OMNIA components. The Zynq consists of a hybrid architecture composed by a dual-core Cortex ARM A9 and a 7-series Xilinx FPGA. The Zynq is an innovative SoC characterized by a powerful ecosystem that greatly simplifies the development process and shrinks the time to market.

The nS-ESD-GW has been designed following a modular approach; this enables the tight partitioning and isolation between internal components involved to implement security, communication and monitoring functions. Furthermore the modularization eases the adaptation process of nS-ESD-GW to the avionic scenario.

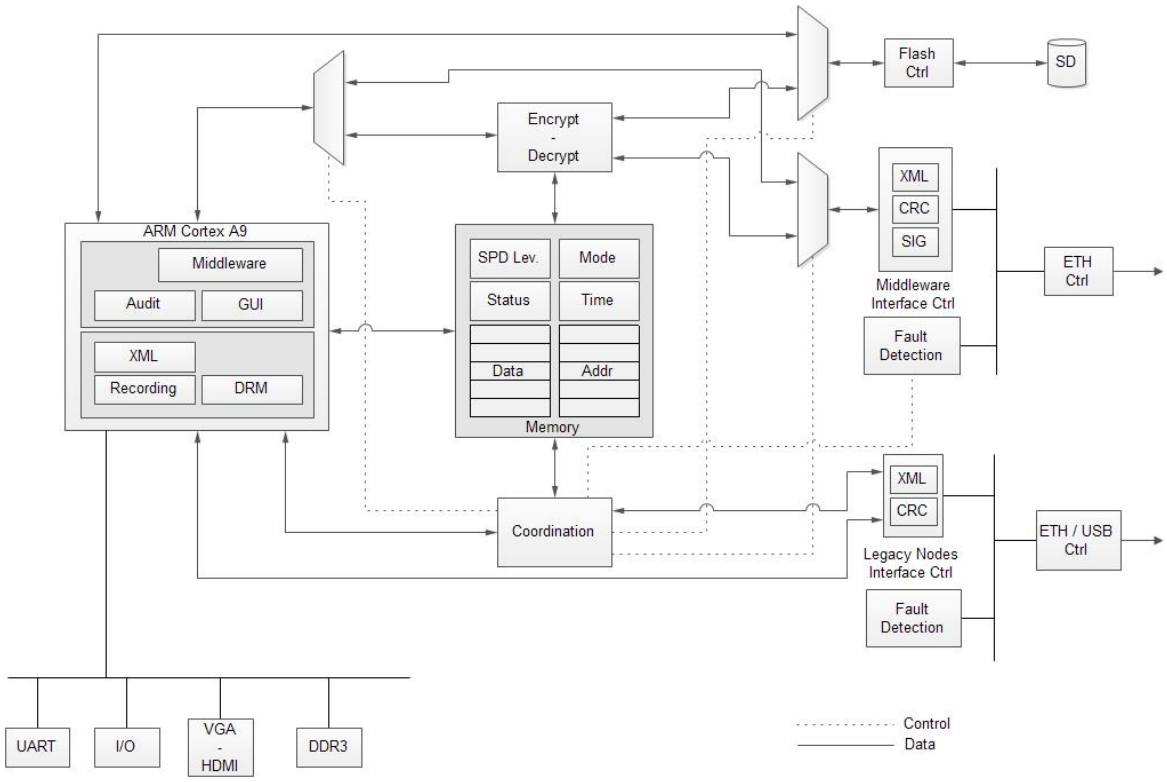

Figure 6-3: nS-ESD-GW HW architecture

As shown in Figure 6-3, the nS-ESD-GW is constituted by several modules hereafter specified:

- **Dual core Cortex ARM A9 in asymmetric multiprocessing (AMP) configuration.** Each processor is configured to run its own software and, in particular, a Linux distribution runs on CPU1 while bare metal applications run, as parallel threads, on CPU2.
- **Encrypt/Decrypt IP core.** This component has been developed as FPGA-based module to assure high flexibility and performances. The presence of this block guarantees the confidentiality and the security of data.
- **Coordination module.** It provides balancing functionality according to the SPD level. To assure the required level of security and dependability, it dynamically adapts its configuration and resources used.
- **Memory.** It stores dynamic data blocks which contain: the status of OMNIA's components, the status of the nS-ESD-GW, SPD levels received by the nSHIELD Middleware, operational mode and freshness information.
- **Middleware Interface Controller.** It is constituted by different sub-components: interfaces, mechanisms of digital signature check, fault detection and the data integrity. In the context of the avionic scenario, it represents the interface between the nS-ESD-GW and the nSHIELD Middleware through the SDR cognitive radio.
- **Legacy nodes Interface Controller.** Similar to the Middleware Interface controller, this component has been subdivided into sub-modules to manage the data integrity, the fault detection and the messages conversion. In the context of avionic demonstrator, it represents the interface between the nS-ESD-GW and the OMNIA platform.
- **Additional peripherals.** The nS-ESD-GW also offers a set of common ready-to-use interfaces as: UART, general purpose I/O, VGA and HDMI.

As previously mentioned, Cortex ARM A9 processors are in AMP configuration; this mechanism allows to run an operative system and bare metal applications with the possibility of loosely coupling those applications via shared resources. Figure 6-4 depicts the software layered architecture designed.

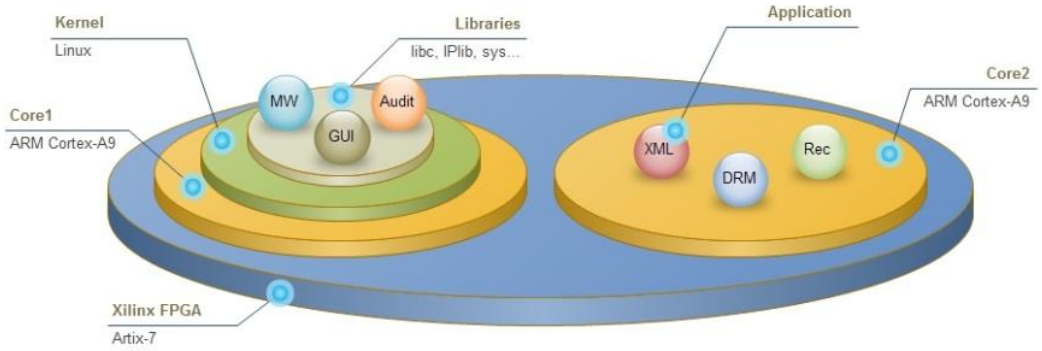

Figure 6-4: nS-ESD-GW SW partitioning

According to this configuration, the Linux operative system is responsible for:

- Audit
- Graphical user interface for monitoring
- nSHIELD Middleware management application

The bare metal applications are responsible for:

- XML parsing and filtering
- Recording / system dump
- Dynamic reconfiguration

6.2.1 n-ESD-GW Gateway SPD features

The nS-ESD-GW is a component defined into the SHIELD framework. Its scope is to foster the interconnection of legacy nodes building up a SHIELD cluster. As constituted, the cluster will inherit from the ns-ESD-GW several SPD features; in particular the cluster will have:

- **Security:** Mechanism for encrypted communication. The cluster nodes will leverage on encryption/decryption methods provided by nS-ESD-GW to exchange messages with other SHIELD components.
- **Security:** Mechanisms for data and message integrity. These mechanisms will ensure the accuracy and the consistency of the exchanged messages.
- **Dependability:** Mechanism for Fault detection. They are obtained by the means of fault tree logic and decision support systems.

Dependability: Mechanism for internal Cluster reconfiguration. Getting information about the nodes status, current faults and context, the system is able to identify a new nodes configuration and eventually apply it to the nodes cluster.

In Figure 6-5 a logical architecture of functionalities provided by the Gateway is outlined.

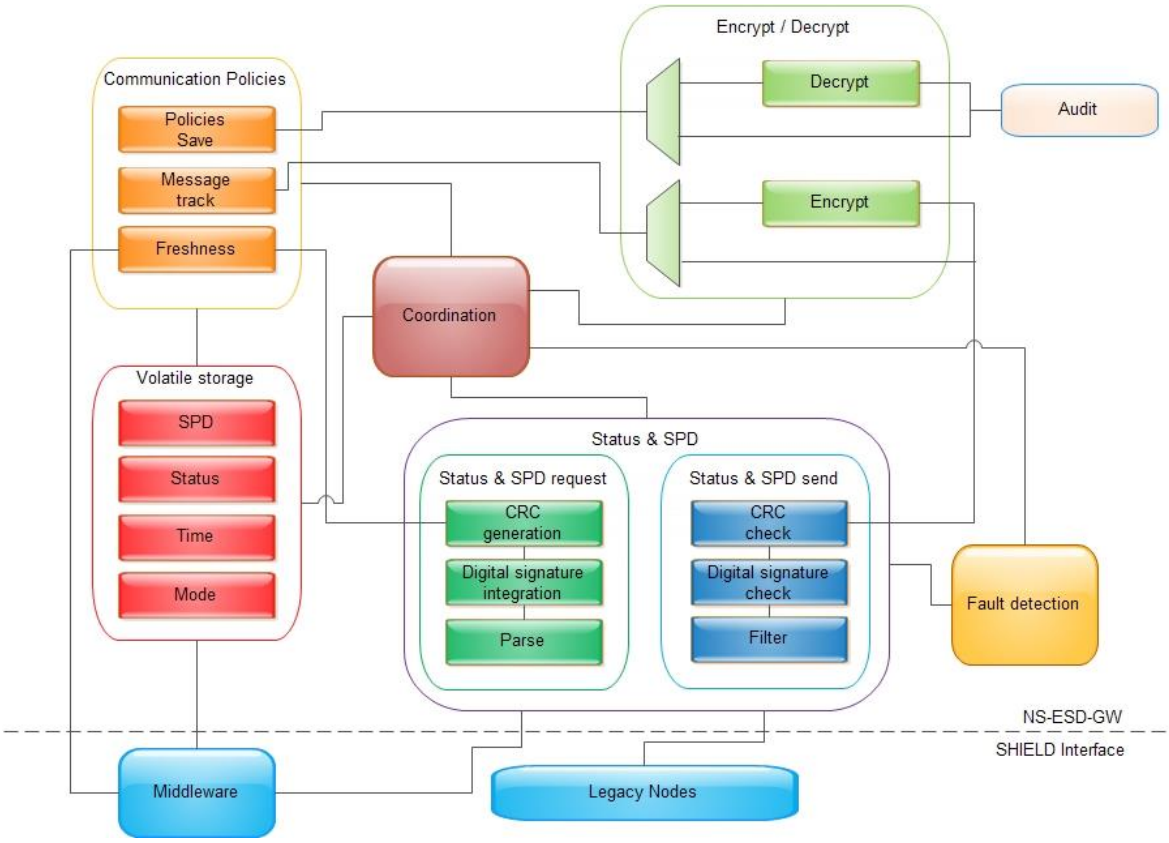

Figure 6-5: nS-ESD-GW Functionalities

The nS-ESD-GW encompasses several communication policies. These policies specify and regulate interrogations (interval, priority, broadcast, unicast, etc), type of messages to be dispatched and/or to be accepted (message alive time, timeout, etc) and the Gateway operational modes. The policies are not upgradeable, avoiding by design the risk of malicious attacks. In accordance to the policies of the nSHIELD framework, the nS-ESD-GW is able to evaluate the SPD level provided by the Middleware and to change its operational mode. This means that the Gateway is able to:

- Increase/decrease the rate of the messages read/write to OMNIA
- Enable/disable cryptographic modules
- Increase/decrease the writing of the logfile concerning the audit function

The nS-ESD-GW encompasses two distinct Data Freshness methods. Stored data can be updated periodically or upon system requests. The update time interval is controlled by an internal configurable register. A dedicated set of registers are used to store information about the data freshness. Specifically, the number representing the amount of time since data have been stored is saved into a specific register of the Gateway and it is available as output.

The confidentiality and the security of private information are ensured through the adoption of encryption/decryption modules for the data writing. A non-volatile memory is used to store long-term data. While the Gateway is running in a secure mode the data processed and algorithms are stored as encrypted. Also, a mechanism of digital signature check is applied during the communication with nodes in order to detect any malicious attempt to open a non-trusted communication channel.

6.2.2 Gateway nS-ESD-GW

Table 6-1 reports requirements covered by the nS-ESD-GW.

Table 6-1: nS-ESD-GW Verification cases

ID	nSHIELD Requirement	Mean of verification	Description
T.1	REQ_ND02 Data Freshness	A	The nS-ESD-GW encompasses two distinct Data Freshness methods. Stored data can be updated periodically or upon system requests. The update time interval is controlled by a configurable parameter. The system is endowed by a set of registers that contain information about the data freshness. [A] – A number representing the amount of time since data have been stored is saved into a specific register of the Gateway and it is available as output.
T.2	REQ_ND03 Digital Signatures	A, T	A mechanism of Digital Signature check is applied to communicate with legacy nodes. [A] – Once a legacy node is elected as trusted, its *status* is written into the data memory. [T] –A non-trusted node, or a node that is not capable of sharing a trusted digital signature, will be detected by the Gateway and any attempt to open a communication channel will refused.
T.3	REQ_ND04 Policy updates	D	The nS-ESD-GW is endowed by several communication policies. These policies specify and regulate the interrogations (interval, priority, broadcast, unicast, etc.), the messages to be dispatched and accepted (message alive time, timeout, etc.) and the Gateway modes. [D] – Policies are not upgradeable, avoiding by design the risk of malicious attacks.
T.4	REQ_ND14 Storage of private information	A, D	The confidentiality and the security of private information is ensured through the adoption of encryption/decryption blocks for the data writing. A non-volatile memory is the support for the storage of long-term information. [A] – During the secure operational mode execution all data that are stored into the memory are encrypted.
T.5	REQ_ND21 Dynamic security behaviour	A	In accordance to the policies of the nSHIELD framework, the nS-ESD-GW is able to evaluate the SPD level provided by the Middleware and to change its operational mode. This means that the Gateway is able to: • Increase/decrease the rate of the messages requests sent to legacy nodes; • Enable/disable cryptographic modules; • Increase/decrease the writing of the log file concerning the audit function. [A] – As consequence of the operational mode changing, log files are stored into the memory with a different rate. Likewise, data saved into the non-volatile support are encrypted if the status of the Gateway requires a more accurate behaviour.

T.6	REQ_ND23 Hardware/Software co-design	D	The platform used to develop the nS-ESD-GW prototype is a chip that integrates hybrid architecture composes by a dual-core ARM Cortex A9 and a 7-Series Xilinx FPGA. This allows the possibility to use a specific design flow that speed up the entire development process and permits the coexistence, on the same device, of hardware-software co-design techniques.
T.7	REQ_ND24 Situational-aware and context-aware SPD	D	With the aim to optimize Gateway's performances, a coordination module is able to provide a services balancing according to the SPD level. [D] – A module that provides fault detection and SPD evaluation encompasses the nSHIELD algorithms. It consists of several software modules and some FPGA-based IP processing blocks.

6.3 SPD-driven Smart Transmission Layer (Prototype 9)

Communication between the UAV and the control center shall be based on utilization of the capabilities of the highly-reconfigurable, computationally unconstrained nSHIELD SDR/Cognitive-capable node, that is, its **Smart Transmission Layer** functionality, developed within the task T4.1.

Role of the Smart Transmission Layer is providing reliable and efficient communications even in critical channel conditions by using adaptive and flexible algorithms for dynamically configuring and adapting various transmission-related parameters. Namely, for the purposes of the Dependable Avionics demonstrator, the following will be exercised and demonstrated:

- Basic network functionalities:
    - network entry;
    - node authentication;
    - internode communication;
    - topology awareness(number of nodes, mutual visibility, connection, location);
    - reconnection of a node after power cycle/link loss;
    - choice of operating frequencies in accordance with the predefined frequency plan
- Waveform and channel interoperability - by using the appropriate software tools, we shall be able to model different kinds of waveforms and emulate different channel conditions
- Jamming detection and counteraction - By measuring link channel quality, and performing consistency checks between SNR-PER and SNR-location, a decision on whether jamming takes place can be taken. In this case, a security counter-algorithm shall be deployed. The algorithm shall encompass the following functionalities (which of them shall be exercised at a given moment depends on the SPD level imposed by the overlay):
    - moving to a new frequency
    - changing physical or logical waveform parameters, i.e. modulation, bit-rate, transmit power, FEC and MAC protocols
    - selecting a different waveform for transceiving

Basic demonstrability of the secure and dependable communication can be depicted by the following scenario:

1. A wireless node-to-node communication between the two communication modules (one is placed on the vehicle and one on the ground as the control center) is initiated on a chosen frequency in VHF/UHF band. SPD level is set to maximum (10), meaning that all transmission parameters,

such as modulation, channel coding, etc., are set to provide the highest resilience to possible interference. Visualization of the link quality is provided on the control center.

2. Jamming disturbances on the channel that is momentarily used for communication are created using another SDR. Jamming power is gradually increased (with SDRs, this is fairly easy to do on-the-fly), until the control center and the onboard-radio decide that the interference level is too high for the normal communication to continue. Link quality is adequately depicted at the control center (possibility of creating some sort of visual/audio “alert” message).
3. Both radios change their operating frequency according to a pre-defined scheme. Non-interfered communication takes place once more, and the satisfying link quality is restored.
4. SPD level is changed (e.g. to 5). Several transmission parameters may now be changed (e.g. higher-order modulation techniques, reducing no. of transmitted redundant bits, etc.), which typically reduce robustness but allow for a higher data rate.
5. Steps 2) and 3) are repeated.

The hardware platform consists of two cooperating entities. Secure Wideband Multi-role – Single-Channel Handheld Radio (SWAVE HH) is used as a RF front end and as the secondary processing platform, whereas OMBRA v2 is used as the primary processing platform. SWAVE HH and OMBRA v2 can be connected either through a high speed serial connection or through a USB/Ethernet, depending on the required throughput. OMBRA v2 needs to be plugged into a carrier board providing electrical interfaces towards radio and external instrumentation and power supply.

Sketch of the complete platform is as follows:

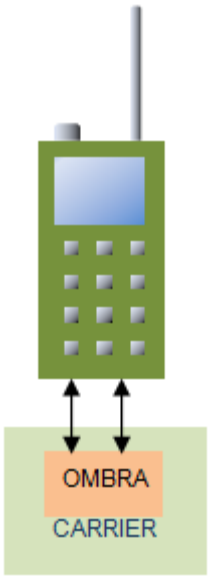

Figure 6-6: Smart transmission layer platform

SWAVE HH is SCA 2.2.2 compatible, and supports reconfiguration of all of its transmission parameters on-the-fly. It is capable of operating in the complete VHF and UHF bands.

OMBRA v2 is a powerful embedded platform equipped with a GPP, DSP and FPGA, being suitable for highly-demanding computational tasks.

More in-depth technical details regarding the SPD-driven Smart Transmission Layer are provided in deliverables D4.2 and D4.3

6.4 Reliable Avionic (Prototype 30)

This field incorporates ***Reliable Avionic Systems*** (No 30), covering the areas: Reliability, Availability, Safety, Confidentiality, Integrity, and Maintainability. Alfatroll has proposed to demonstrate how these ***extemely demanding objectives*** can be obtained by using its own, Knowledge Based Technology, IQ_Engine. This has been approved at SES July 4th 2013 (Massimo Traverzone, Silvia Larghi, Tor O Steine). The demo setup involves*:*

1. Using ordinary PC(s) for computer platform(s). Ordinary Windows laptops used. A demo will probably be set up using one PC for Ground Control Station, and another to simulate the UAV(s).
2. Use any Ground Station suitable for the job. MAV GCS and/or QGCS are used. Both are tested.
3. Use any link. MAVlink has been selected. Can run both fixed wing and rotational wing, single or multirotor. To be replaced by Omnia in a full implementation. This will not affect the basic properties of the demonstrator much.
4. Use any flight simulator. We have selected a large fixed wing «UAV» for the purpose.
5. We chose to use IQ_Engine for all UAV on-board functions, albeit it is possible to use an existing Autopilot and control it via the IQ_Engine Cognitive pilot.

The following image illustrates the final set-up of communications systems and IQ_Engine.

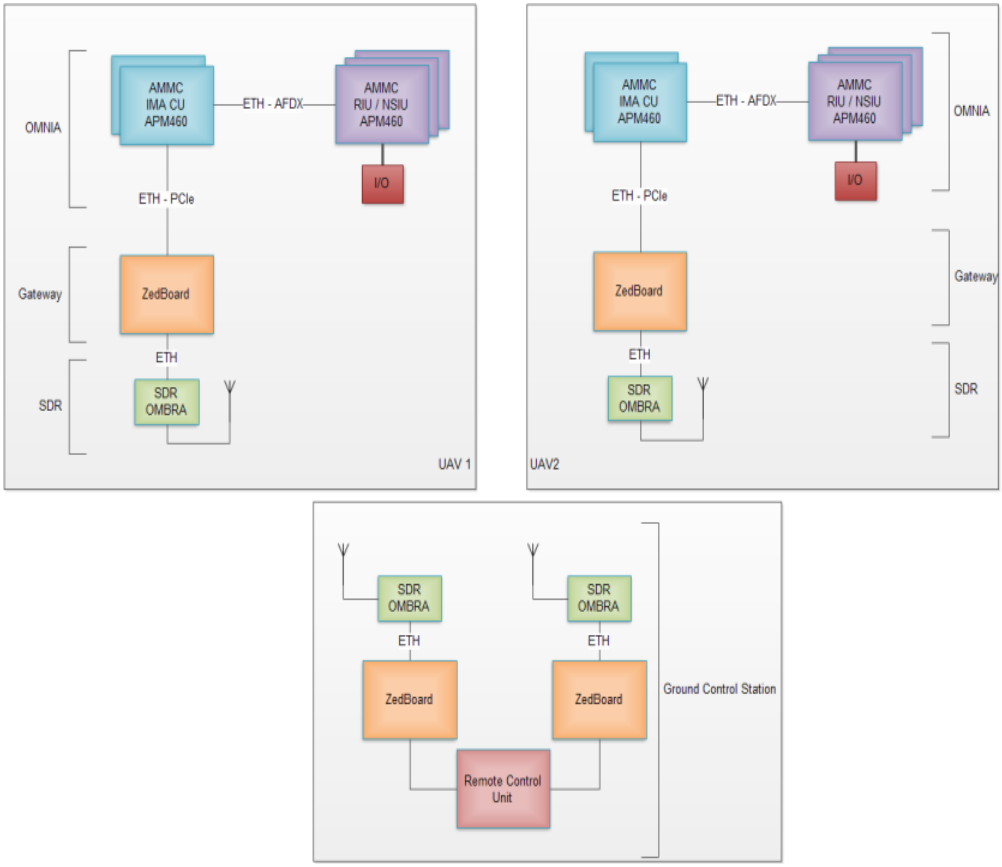

Figure 6-7: Basic Omnia framework for IQ_Engine demo

This is not the case at the current stage. Instead, the demo set-up involves the following components. They include the Silent Wings flight simulator, flying a fixed wing motorglider (approximately the size of a Predator), a MAVGCS Ground Control Station for Mission Planning and operational control, and the IQ_Engine (not shown), controlling the simulator.

We suggest that the demo and test are split in two parts:

1. Demo and test of the above framework
2. A separate test of IQ_Engine and the application scenario.

Thus, this extremely ambitious task of demonstrating “Reliable Avionics System” can be performed on a low budget (It may be worth reminding that the JSF fighter program seem to be exceeding 30 million lines of code, and EADS next generation U(C)AV are indicated to include 65 million lines of on-board code). A fully fledged demo of a Reliable Avionics System within the nSHIELD budget framework clearly is impossible. Yet, Alfatroll claims that we can demonstrate a principle that can help reduce the complexity of the on-board avionic systems considerably, thereby demonstrating how reliable systems can be obtained while keeping the costs low.

For the demo, Alfatroll has chosen a solution where any airport covered by Silent Wings can be used (including some in the Alps). We have chosen Notodden airport in Norway, since it is known to the Alfatroll people, and it has a variety of mountainous terrain. The simulator can be influenced by both wind and turbulence on a scale from NONE to STRONG:

Figure 6-8: IQ_Engine test set-up

The application scenario we are working towards is this:

1. UAV#1 loitering above Target (fixed or moving), monitoring and transmitting video to the ground.
2. UAV#1 fails, reports error, and returns to base.
3. UAV#2 takes over the tasks that were covered by UAV#1, as commanded by the GCS.
4. All the above involves Multi-UAV Ground Control Station, two instances of UAV, and the functionality described. All while being true to the goals (to the left).

All this while maintaining: Reliability, Availability, Safety, Confidentiality, Integrity, and Maintainability.

The current status is this:

System development started September 15th (from partly finished prototype, after substantial planning and preparations).

- We are on schedule.

Main points proven:

- Full functionality possible
- Compact code & Knowledge base
- Safe and reliable operation
- Almost all maintenance in the database, no complex software maintenance

The remaining work is scheduled and properly manned. Due to a limited budget, we are not anticipating to implement the above in a real flying UAV, but we intend to demonstrate the functionality in a variety of types of flying (simulated) platforms. The on-board software systems will be refined to demonstrate how complex functions can be controlled by very compact software, with the functionality mostly reside in a Knowledge Database. This is, according to Alfatroll, necessary in order to achieve the nSHIELD goals of Reliable Avionic Systems architecture.

6.4.1 Areas of functionality to cover:

In order to demonstrate how IQ_Engine can contribute significantly to better reliability in avionics systems, we intend to demonstrate some or most of these functions in the project:

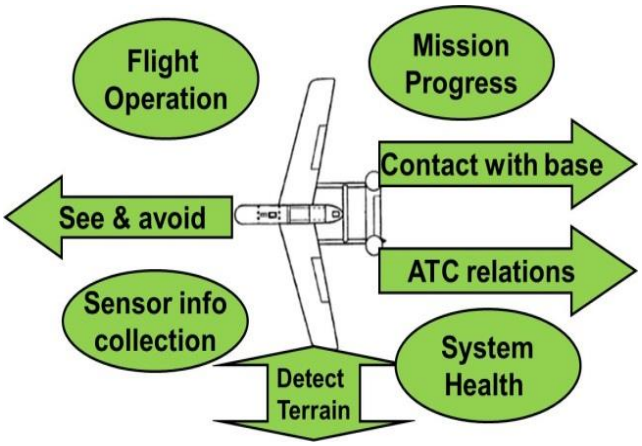

Figure 6-9: Application areas that can be covered by IQ_Engine Cognitive Pilot

We are already working on a, b, c, of the listed tasks below, but will extend the list as far as possible within the budget of the nSHIELD project.

a. **Flight Operation** are all normal tasks covered by an autopilot, such as bringing the plane safely from waypoint to waypoint along the scheduled trajectory and at the designated time schedule (dubbed 4D control)

b. **Mission Progress,** including necessary actions to be carried out at pre-defined locations or in certain situations. The task includes contingency management, i.e. reactions to unexpected events that may occur during the flight, such as icing, low on fuel, etc.

c. **Contact with base,** including reports back in the form of data streams and status messages, as well as the ability to react promptly to orders from the ground station.

d. **ATC relations.** In the beginning, all communication with air traffic controllers (ATC) will most likely be through the ground station controller. When ATC communications becomes message based (Single European Sky specifies this), more and more can be handled directly by the unmanned aircraft itself. Proper responses and reactions are expected by the on-board intelligent controller, of course.

e. **System Health.** A major part of the responsibility of an autonomous system is to keep track of its own operational status. System health can include such tasks as e.g.: Tank/Batteries – remaining range compared to remaining mission, Icing on wings or sensors, Control responses, Electronic systems status (temp, humidity..), Sensor status (visibility,…), Radio link quality, Engine *status.*

f. **Detect Terrain.** By following the plane's movements in relation the the terrain below, the control logic shall be able to avoid conflicts with terrain objects, controlled or prohibited airspace, tall objects on the ground, etc. The plane shall also be able to react differently to corrective actions at low level flight from normal high level flights.

g. **Sensor info and collection.** Sensor information is normally collected on-board or forwarded to the ground control. If the sensors detect items of interest to the current mission (e.g. a hot spot in the sea during search and rescue), this may cause the control logic to launch specific actions (e.g. drop marker, go lower and take picture/video).

h. **See and Avoid.** This is the focus of many UAV producers at the time: to make systems that can detect other air traffic or other obstacles in the same, non-segregated airspace, and react as if a pilot was on-board. Using ADS-B and other sources, keeping track with traffic nearby is possible. But visual and radar detection will always be less precise. For the project, we are going to receive feed from http://www.flightradar24.com/33.59,35.78/7, place ourselves in an area with heavy traffic, and let the IQ_Engine react as required according to safe detect and avoid, and the (ICAO Annex 2) rules of the air.

Feed from FlightRadar around Rome looks e.g. like this:

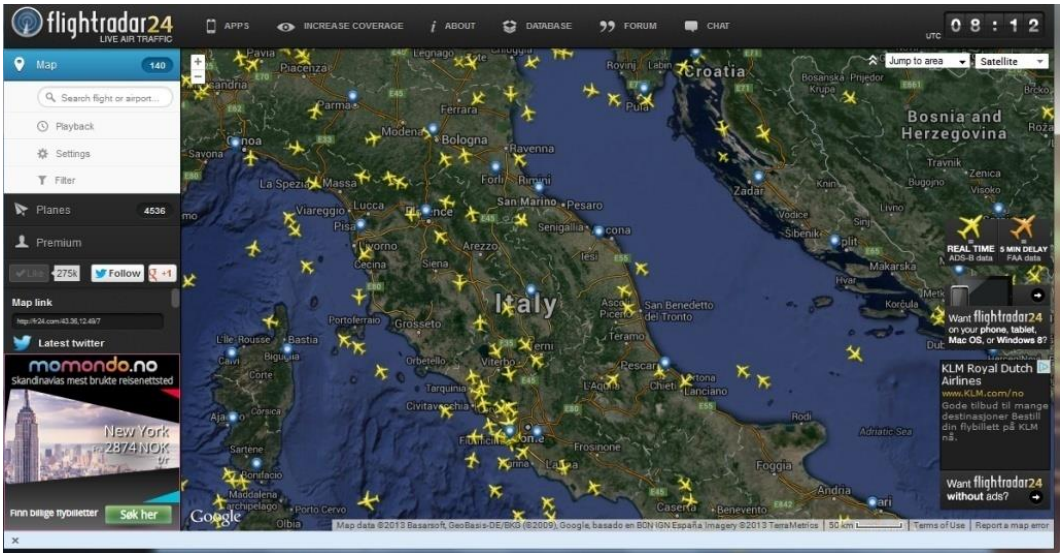

Figure 6-10: Feed from FlightRadar, for demo of Detect&Avoid

Final comments:

If taken literally, the task of demonstrating "Reliable Avionics Systems" on a very limited budget within the nSHIELD framework is next to impossible. We have, in cooperation with SES, chosen a middle road solution which will focus on HOW reliable avionic systems can be achieved, by demonstrating this in a small scale. The solution proposed include introducing a new technology which opens for hereto unknown properties within software development for avionics.

Alfatroll claims this is the way to achieve the intended goals in nSHIELD.

6.5 Semantic model (Prototype 26)

Also for the avionic demonstrator, specific semantic models will be instantiated, following the procedures defined in D5.3. As already described, the integration of this prototype in the common platform is done in two ways:

i. By providing the components' responsible with the "guidelines" to write down the Ontology model for their component as well as the Domain Dependent Library

ii. By codifying this ontology into an xml file that can be parsed by the OSGI to extrapolate relevant information.

6.6 Metrics (Prototype 27)

nSHIELD proposes 2 types of metric aggregation measurement. Both types of metrics are described in document D2.8 Final Metric Specification.: Attack Surface Metric and Multi Metric approach. Both need an integration procedure with respect the holistic nSHIELD platform.

This integration approach will be held by incorporating the aggregation formula to OSGI Middleware nSHIELD Overlay, and in particular by embedding it in the semantic model used in the SHIELD framework. This middleware will enable a container for aggregation formula and/or algorithm.

However it must be understood that both approaches have to be tuned by operator experts, so that integration will be finished with both perspective: this one which will be automatically addressed and one more manual one with the opinion of experts.

6.7 OSGI Middleware (Prototype 25)

Also for the avionic demonstrator, the OSGI framework is adopted to implement the SHIELD Middleware (see section 4.8 for additional details on the Knopflerfish implementation adopted in the project).

On a deployment point of view, this framework is installed into a Notebook that is supposed to be part of the Remote Control Unit. This PC is interfaced directly with the Intrusion Detection Bundle and consequently with the Gateway and the rest of the avionic demonstrator; the first interconnection hop is through a network (most likely an Ethernet LAN).

The interfaces with the Secure Discovery Bundle (in charge of populating the service databases) and the Security Agent are internal and implemented directly in Java Language.

Additional interfaces will be implemented to allow seamless integration with the Reliable Avionic System that will be used as an SPD functionality to be managed by the Overlay (in particular a functionality improving the reliability of the overall system).

6.8 Control Algorithms (Prototype 20)

Also for the avionic demonstrator, the SHIELD control algorithms are the simplest prototype to be integrated in the common platform, since they are embedded in the Middleware code and in particular they are implemented in the Security Agent bundle. On a practical point of view, the control algorithms will be a set of software instructions executed by the OSGI.

Also in this case this solution will allow decoupling between the control algorithm and the implementation of the control action on the system (that is in charge to the Security Agent). Moreover, in case the OSGI libraries will not be suitable to solve the composition (mathematical) problem, then the support of external computational software tool can be foreseen, like, for example, Matlab, that could be easily called by the Security Agent routines to solve the problems.

6.9 Middleware Intrusion Detection System (Prototype 22)

6.9.1 IDS prototype interfaces

In its current status, the preliminary IDS prototype has generic network interfaces for receiving and forwarding requests that are to be filtered. In Figure 6-11, these bi-directional interfaces are denoted as **IF-2**. The IDS prototype will receive and optionally forward messages without altering their content or re-encapsulating them. In this sense, **IF-2** is a homogeneous interface between the Middleware services and the prototypes which use them.

It is however anticipated that TAP / TUN virtual network interfaces could be used to physically separate and protect internal (Middleware services) and external (other components and networks besides Middleware) network domains. These changes could mostly be implemented in a transparent manner for the system components using middleware services, but may impact how connection methods towards middleware services should be implemented. The design of the network domains and the connection methods used will be studied at the time of integration with other Middleware components.

Using the IDS prototype requires setting up network infrastructure so that requests are received by the gateway instead of the middleware services natively. For this purpose, the Intrusion Detection and Filtering Module provides additional function call interfaces towards the Middleware services that implement the use of the IDS – see **IF-3** in Figure 6-11. At the time of integration, it needs to be determined which Middleware services are to be protected against DoS/DDoS attacks, and which operation mode of intrusion detection (blacklisting / whitelisting) is more beneficial to be used for each. Use of IDS prototype for these services is straightforward: by adding a few lines of Java code to the services, they can be enabled to use intrusion detection functionality.

The nSHIELD Overlay functionality will be responsible to monitor SPD properties and control desired SPD level for the prototypes. The function interface IF-3 provides all functions for this purpose; for more details, see next chapter.

Further information and source code for the IDS prototype is available in D5.2 [10] and D5.3 [11].

6.10 Interactions map

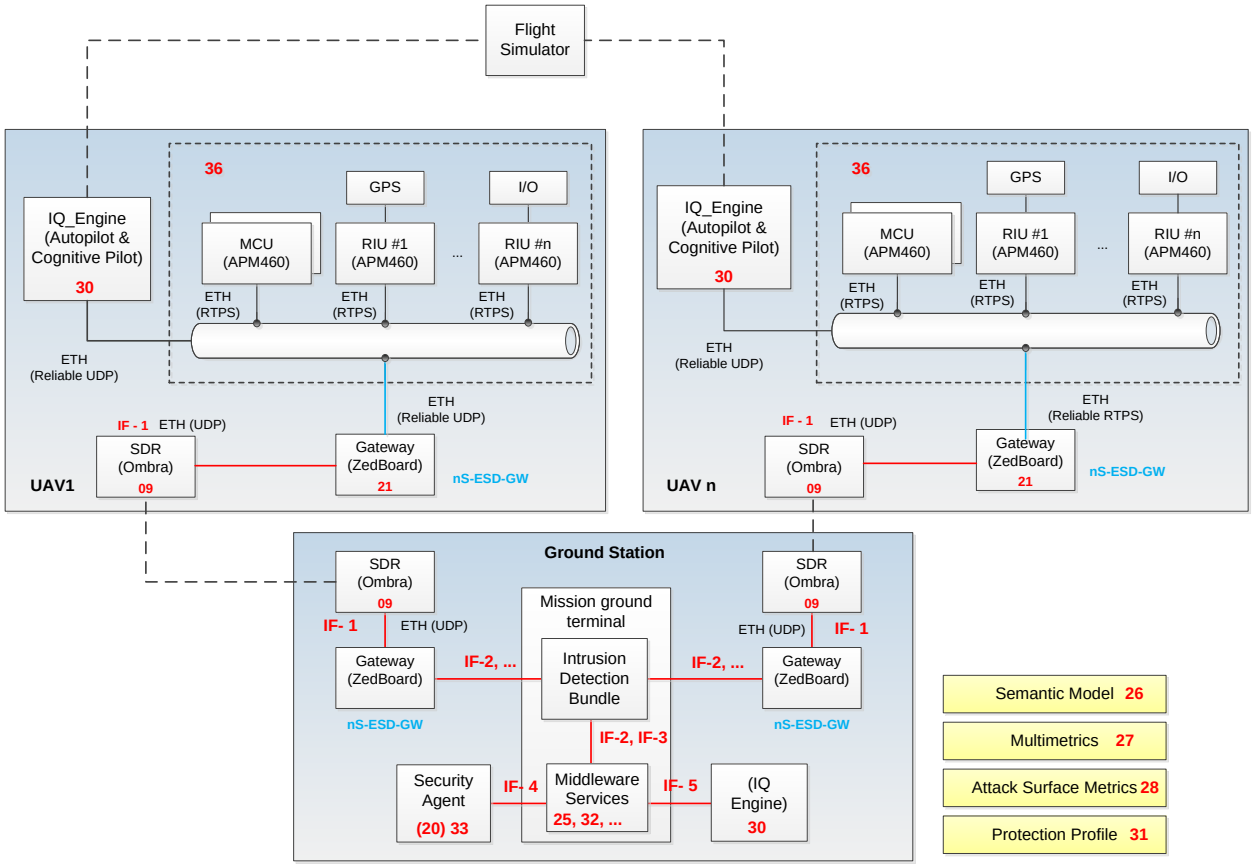

Figure 6-11: Dependable Avionic scenario interactions

7 Components of the General nSHIELD Framework

As explained in the introduction of this document, the integration methodology is structured on the distribution and usefulness of individual prototypes in each scenario. The objective is to integrate components in a common platform, after their validation and verification as standalone and collaborative objects. The architecture of a common platform, consisting of tens of prototypes necessary to cover different SPD requirements and levels, introduces designing generalities. Subsequently, there are nSHIELD components not directly correspondent to one of the very specific use cases developed in the scenarios. Additionally, there are prototypes matching the criteria and requirements of more than one scenarios and therefore could be listed under the general nSHIELD prototype framework. This section covers the description of these categories of prototypes.

7.1 Link Layer Security Prototype (Prototype 23)

Temperature, humidity and other kind of sensors are used on multiple situations to provide information about the surrounding environment.

Sometimes this information is sensible and the link layer should provide authentication and integrity to preserve data.

In order to show this features the prototype will communicate a mote with a base station where information could be processed.

7.1.1 Link layer prototype interfaces

The link layer prototype developed could be connected to other systems through a gateway where network layer security will be provided.

7.1.2 Link layer prototype SPD features

This nSHIELD link layer prototype is composed of several sensors which ensure their communications thought CTR, CBC-MAC or CCM algorithms.

The main functionalities of this prototype are:

a) **RF Module** that supports the 802.15.4, based on the CC2420 that provides a wireless communication link.
b) **Authentication and confidentiality hardware capability**

With this prototype some SPD functionalities that could be covered are listed below:

Table 7-1: Link Layer SPD features

Authentication	Applying CCM or CBC-MAC the receiver could be secure that the data received is provided by the correct transmitter
Confidentiality	Applying CCM or CTR the data received will be encrypted and only if the key is shared between RX and TX the data could be correctly decrypted.

7.1.3 Link layer prototype environment

For the use case demonstration, the link layer control system can be integrated with other nSHIELD components through a base station where data will be received.

7.2 Protection Profile (Prototype 31)

The nSHIELD project has the ambition to be a commercial standard for Security, Privacy and Dependability regarding embedded systems. At this purpose the idea of a Protection Profile (at the moment only for middleware layer) is a first step to define a security problem definition and security objectives for embedded systems.

As defined in D5.3 [11], a protection profile (PP) is a Common Criteria (CC) term for defining an implementation-independent set of security requirements and objectives for a category of products, which meet similar consumer needs for IT security. Examples are PP for application-level firewall and intrusion detection system. PP answers the question of "what I want or need" from the point of view of various parties. It could be written by a user group to specify their IT security needs. It could also be used as a guideline to assist them in procuring the right product or systems that suits best in their environment. Vendors who wish to address their customers' requirements formally could also write PP. In this case, the vendors would work closely with their key customers to understand their IT security requirements to be translated into a PP. A government can translate specific security requirements through a PP. This usually is to address the requirements for a class of security products like firewalls and to set a standard for the particular product type.

Considering this PP definition it is evident that it is a particular type of prototype which is completely divorced from the speech of the integration of the prototypes. On the contrary, it makes the rules or rather the SPD requirements that must be met by prototypes Integration that make up an embedded system aiming to be SHIELD compliant (as indicated above, at this time are shown only the SPD requirements that the middleware of the system must meet).

7.3 Attack Surface Metrics (Prototype 28)

Attach Surface Metric approach starts from the following considerations:

1. A threat is the origin of the fault chain (fault -> errors -> failures) for the dependability concerns and as the potential for abuse of protected assets by the system for security concerns.
2. The attacker is the threat agent; it is a malicious human activity or non malicious event.
3. An attacker uses nSHIELD's entry and exit points to attack the system.

So it was introduced an entry and exit point framework to identify three relevant factors: Porosity, Controls, and Limitations.

An entry and exit point contribution to the attack surface reflects factors' likelihood of being used in attacks. For example an entry point running a method with root privilege is more likely to be used in attacks than a method running with non-root privilege. We introduce the notion of a damage potential-effort ratio (der) to estimate porosity contribution.

A system's attack surface measurement (Actual SPD Level) is the total contribution of the system's factors along the porosity, controls, and limitation.

Each supplier of a product or system that will be part of this demonstrator must provide the data needed for the calculation of SPD level defined by the adopted metric approach.

These data will be provided by filling in an excel sheet which is being finalized and will contain all the information necessary to Actual SPD level calculation.

All data collected will then be given as input to the middleware of the system which will be able to process them in order to provide, in a dynamic way, the variable parameters values of the implemented security features (controls) the values in order to allow the system to reach the desired "Actual SPD Level".

The Attack surface metric approach definition and the details of data to be provided are contained in deliverable D2.5 [10].

7.4 Key Exchange Protocol (Prototype 02)

Integrating the Control Randomness Protocol requires interfacing with both the lightweight cryptographic framework developed in the premises of the nSHIELD architecture as well as the network layer security prototype.

Since CRP does not dictate how all the cryptographic keys are transferred to the receiver, the initial transfer of all the keys is handled by the chosen public key cryptography scheme dictated by the network security layer. The CRP dictates how all these keys are used and reused within a time frame composed of many conventional sessions.

The underlying implementation of the CRP relies on the usage of a symmetric cryptographic algorithm as well as a keyed hashing algorithm. During the implementation phase of the prototype, AES and HMAC-SHA256 were used as reference algorithms. The actual choice of algorithms depends on each scenario and on the availability of ciphers in the underlying cryptographic framework. The underlying framework declares the availability of ciphers and the CRP implementation chooses based on the ranking of each cipher in the list.

7.5 Recognizing Denial of Service (Prototype 13)

The DoS attack detection mechanism involves cooperation between components belonging to all three layers on the nSHIELD architecture. However, the principal algorithmic operation is considered to be a network process and is therefore described in the network related documents. The scheme can be seen as an algorithmic operation which is fed with inputs from various components and provides a set of results relating to the identification of a DoS attack that is underway.

7.5.1 Interfaces

In order for this prototype to be integrated, interfaces exist to communicate with appropriate processes. The DoS attack defence prototype can be mainly considered to lie in the network layer and therefore provides a network service. The various components of the software prototype need to be able to communicate with components providing input information. These include interfaces for communication with the Power unit module, OS calls for CPU monitoring information and access to the exchanged traffic at the network protocol level.

7.5.2 Environment

The prototype was designed to be consisted of standard C/C++ libraries and therefore can be integrated in all operating systems and environments.

7.6 Adaptation of Legacy Systems (Prototype 29)

7.6.1 Prototype interfaces

The software prototype of Adaptation Of Legacy Systems provides a mechanism that allows legacy devices to be integrated into nSHIELD framework and make use of nSHIELD services. This is done by using specific software adapters (OSGi bundles) that contain the semantic information necessary for the discovery/composition procedure and that are able to communicate them.

The ad-hoc software is OSGi bundles in the nSHIELD side and in L-ESD side. The software in L-ESD side provides discovering of the nSHIELD remote services and the software in nSHIELD side provides advertising nSHIELD services for being remotely discovered.

Using this prototype for the nSHIELD services requires make use of R-OSGi bundle in both sides and in nSHIELD side adding a few lines of Java code to the services so they can be advertised and being remotely accessed.

7.6.2 Prototype environment

This prototype was designed to become part of the Middleware services, thus the software routines for each service to be used remotely by the Legacy Nodes are Knopflerfish OSGI Bundles.

8 Conclusions

A first step of the integration process which is the object under investigation in this deliverable is the systematic collection of all the prototypes/technologies developed throughout the project activities. At this stage some prototypes are working as an initial proof of concept of a given technology running independently from others and able to be verified through individual testing. The process of integrating the large number of different prototypes in a unified framework is a challenging task due to their heterogeneity and the complexity which is increased as the number of technologies increases. Another difficulty related to source code integration is that parts of code are proprietary to protect business activities of the involved partners.

Starting from Chapter 3 the roadmap to construct a framework able to compose systems using different SPD components while addressing functional and SPD application requirements is presented. This tool which must also be architectural compliant and able to provide a SPD level assessment at every stage of application operation will be further refined in the next version of this deliverable. Technologies needed to be integrated to satisfy requirements set for railway security, people identification at the stadium and reliable avionic have been identified and the interconnectivity among different components is presented in Chapters 4-6. The use of the framework to construct all nSHIELD application demonstrators will be the final goal of activities included in this task.

9 References

[1] S. McConell, “Code Complete: A Practical Handbook of Software Construction”, 2nd Edition, Microsoft Press, 2004.

[2] G. Myers, “The Art of Software Testing”, John Wiley, 1979.

[3] nSHIELD Technical Annex v2.4

[4] nSHIELD D2.2: Preliminary System Requirements and Specifications

[5] nSHIELD D2.5: Preliminary SPD Metric Specification

[6] nSHIELD D2.4: Reference System Architecture Design

[7] nSHIELD D3.3: Preliminary SPD Node technologies prototype Report

[8] nSHIELD D4.2: Preliminary SPD Network technologies prototype

[9] nSHIELD D4.3: Preliminary SPD Network technologies prototype Report

[10] nSHIELD, D5.2: Preliminary SPD Middleware and Overlay technologies prototype

[11] nSHIELD, D5.3: Preliminary SPD Middleware and Overlay technologies prototype Report

[12] Atta official web site: https://bitbucket.org/atta-all (2013)

[13] nSHIELD, D3.2: Preliminary SPD Node technologies prototype

[14] Deepak Tagra, Musfiq Rahman, Srinivas Sampalli: Technique for Preventing DoS Attacks on RFID Systems. In 2010 International Conference on Software, Telecommunitaions and Computer Networks (SoftCOM), pages 6-10. (2010)

[15] http://subversion.apache.org

[16] https://wiki.debian.org/Smartcards.